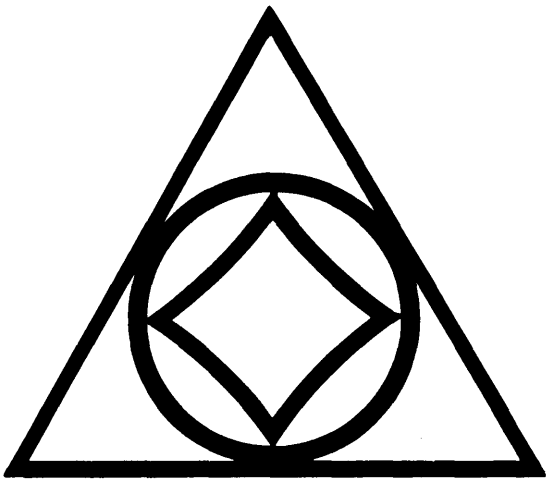


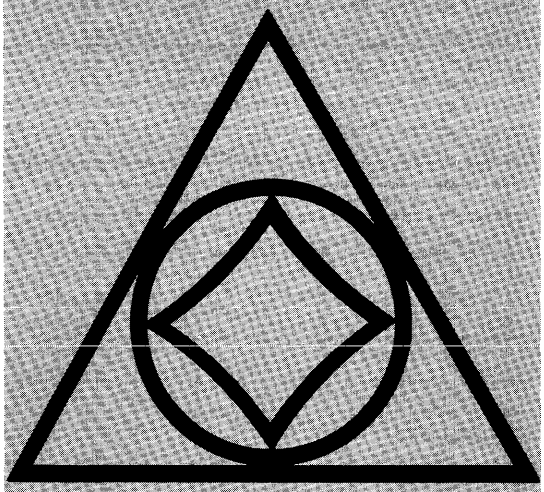
AFIPS

CONFERENCE
PROCEEDINGS

VOLUME 30

1967

SPRING JOINT
COMPUTER
CONFERENCE



AFIPS

CONFERENCE PROCEEDINGS

VOLUME 30

1967

SPRING JOINT COMPUTER CONFERENCE

APRIL 18–20

ATLANTIC CITY, NEW JERSEY

THOMPSON BOOKS
Washington, D. C.

•

ACADEMIC PRESS
London

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1967 Spring Joint Computer Conference Committee or the American Federation of Information Processing Societies.

Library of Congress Catalog Card Number 55-44701
Thompson Book Co.
National Press Building
Washington, D. C. 20004

Sole distributors throughout the entire
world (except North and South America):

Academic Press
Berkeley Square House
Berkeley Square, London, W. 1

CONTENTS

DYNAMIC ALLOCATION OF COMPUTING RESOURCES

A resource allocation scheme for multi-user on-line operation of a small computer . . . 1 — *Allen Reiter*

Effects of scheduling on file memory operations . . . 9 — *Peter J. Denning*

Address mapping and the control of access in an interactive computer . . . 23 — *David C. Evans, Jean Yves Leclerc*

MANAGING THE DEVELOPMENT OF COMPUTER PROGRAMS - A USER'S VIEWPOINT

The Air Force computer program acquisition concept . . . 33 — *Milton V. Ratynski*

Configuration management of computer programs by the Air Force: Principles and documentation . . . 45 — *Lloyd V. Searle, George Neil*

The technical specification - Key to management control of computer programming . . . 51 — *Burt H. Liebowitz*

Air Force concepts for the technical control and design verification of computer programs . . . 61 — *M. S. Piligian, Lt. J. L. Pokorney*

COMPUTER LOGIC AND ORGANIZATION

UNIVAC 1108 multiprocessor system . . . 67 — *D. C. Stanga*

Considerations in block-oriented systems design . . . 75 — *D. H. Gibson*

Intrinsic multiprocessing . . . 81 — *Richard A. Aschenbrenner, Michael Flynn, George A. Robinson*

The association-storing processor . . . 87 — *D. A. Savitt, H. H. Love, Jr., R. E. Troop*

VISUAL OUTPUT RECORDING

Digitized photographs for illustrated computer output . . . 103 — *Richard W. Conn*

Mathematical techniques for improving hardware accuracy . . . 107 — *C. J. Walter*

A new printing principle . . . 121 — *W. H. Puterbaugh, S. P. Emmons*

APPLICATIONS OF ANALOG AND HYBRID COMPUTERS

A time delay compensation technique for hybrid flight simulators . . . 125 — *V. Wayne Ingalls*

Backward time analog computer solutions of optimum control problems . . . 133 — *Max D. Anderson, Someshwar C. Gupta*

Cartoon animation by an electronic computer . . . 141 — *Takeo Miura, Junzo Iwata, Junji Tsuda*

Stochastic Computing . . . 149 — *B. R. Gains*

DATA MANAGEMENT

The people problem: Computers can help . . . 157 — *E. R. Keller, II, S. D. Bedrosian*

File handling at Cambridge University . . . 163 — *D. W. Barron, A. G. Fraser, D. F. Hartley, B. Landy, R. M. Needham*

GIM-1, a generalized information management language and computer system . . . 169 — *Donald B. Nelson, Richard A. Pick, Kenton B. Andrews*

Inter-program communications, program string structures and buffer files . . . . . 175 *E. Morenoff*, *J. B. McLean*

DM-1, a generalized data management system . . . . . 185 *Paul J. Dixon*, *Jerome D. Sable*

File management on a small computer . . . . . 199 *Gilbert P. Steil, Jr.*

HANDLING THE GROWTH BY DEFINITION OF MECHANICAL LANGUAGES - A SPECIAL TUTORIAL SESSION . . . . . 213 *Saul Gorn*

I/O DEVICES

An optical peripheral memory system . . . . . 227 *R. L. Libby*, *R. S. Marcus*, *L. B. Stallard*

A rugges militarily fieldable mass store (mem-brain file) . . . . . 239 *W. A. Farrand*, *R. B. Horsfall*, *N. E. Marcum*, *W. D. Williams*

The "Cluster" - Four tape stations in a single package . . . . . 245 *J. T. Gardiner*

BIOMEDICAL COMPUTER APPLICATIONS

The oscillating vein . . . . . 253 *Augusto H. Moreno*, *Louis D. Gold*

Computer applications in biomedical electronics - Pattern recognition studies . . . . . 257 *A. J. Welch*, *Joseph L. Devine, Jr.*, *Robert G. Loudon*

Experimental investigation of large multilayer linear discriminators . . . . . 265 *W. S. Holmes*, *C. E. Phillips*

Effect of stenosis of the blood flow through an artery . . . . . 273 *Peter M. Guida*, *Louis D. Gold*, *S. W. Moore*

SECURITY AND PRIVACY IN COMPUTER SYSTEMS . . . . . 279 *Willis H. Ware*

Security consideration in a multi-programmed computer system . . . . 283 *Bernard Peters*

Security and privacy: Similarities and differences . . . . . 287 *Willis H. Ware*

System implications of information privacy . . . . . 291 *Harold E. Petersen*, *Rein Turn*

PANEL DISCUSSION: PRACTICAL SOLUTIONS TO THE PRIVACY PROBLEM

The ABC time-sharing system . . . . . 301 *James D. Babcock*

Project MAC time-sharing system . . . . . 303 *Edward L. Glaser*

COMPUTING ALGORITHMS

The influence of machine design on numerical algorithms . . . . . 305 *W. J. Cody*

Base conversion mappings . . . . . 311 *David W. Matula*

Accurate solution of linear algebraic systems - A survey . . . . . 321 *Cleve B. Moler*

Recursive techniques in problem solving . . . . . 325 *A. Jay Goldstein*

Statistical validation of mathematical computer routines . . . . . 331 *Carl Hammer*

MACROMODULAR COMPUTER SYSTEMS . . . . . 335 *Wesley A. Clark*

A functional description of macromodules . . . . . 337 *Severo M. Ornstein*, *Mishell J. Stucki*, *Wesley A. Clark*

Local design of macromodules . . . . . 357 *Mishell J. Stucki*, *Severo M. Ornstein*, *Wesley A. Clark*

Engineering design of macromodules . . . . . 365 *Asher S. Blum*, *Thomas J. Chaney*, *Richard E. Olsen*

A macromodular systems simulator (MS2) . . . . . 371 *Richard A. Dammkoehler*

A macromodular meta machine . . . . . 377 *William E. Ball*

The CHASM: A macromodular computer for analyzing neuron models . . . . . 393 *Charles E. Molnar*, *Severo M. Ornstein*, *Antharvedi Anne*

SOME ASPECTS OF COMPUTER ASSISTED INSTRUCTION
Requirements for a student subject matter interface . . . . . 403 *Kenneth Wodtke*
Central vs. decentralized computer assisted instruction systems . . . . 413 *M. Gelman*
Reflections on the design of a CAI operating system. . . . . 419 *E. N. Adams*

INFORMATION PROCESSING IN THE BUSINESS ENVIRONMENT
Nature and detection of errors in production data collection . . . . . 425 *William A. Smith, Jr.*
Serving the needs of the information retrieval user . . . . . 429 *C. Allen Merritt*
The Ohio Bell Business information system . . . . . 433 *E. K. McCoy*

TECHNIQUES IN PROGRAMMING LANGUAGE - PART I
Programming by questionnaire. . . . . 441 *Allen S. Ginsberg*
Harry M. Markowitz
Paula M. Oldfather

Compiler generation using formal specification of procedure-oriented and machine languages. . . . . 447 *Philip Gilbert*
William G. McLellan

An experimental general purpose compiler. . . . . 457 *R. S. Bandat*
R. L. Wilkins

THE BEST APPROACH TO LARGE COMPUTING CAPABILITY - A DEBATE. . . . . 463 *Gerhard L. Hollander*

Achieving large computing capabilities through aggregation of conventional system elements . . . . . 467 *George P. West*

Achieving large computing capabilities through associative parallel processing . . . . . 471 *Richard H. Fuller*

Achieving large computing capabilities through an array computer . . . 477 *Daniel L. Slotnick*

Achieving large computing capabilities through the single-processor approach . . . . . 483 *Gene M. Amdahl*

THE EXPANDING ROLES OF ANALOG AND HYBRID COMPUTERS IN EDUCATION: A PANEL DISCUSSION
"Position" Papers . . . . . 487 *Dr. Lawrence E. Burkhart*
Dr. Ladis E. Kovach
A. I. Katz
Dr. Myron L. Corrin
Dr. Avrum Soudack
Dr. Robert Kohr

LOGIC-IN-MEMORY
DTPL push down list memory . . . . . 491 *R. J. Spain*
M. J. Marino
H. I. Jauvtis

An integrated MOS transistor associative memory with 100ns cycle time . . . . . 499 *Ryo Igarashi*
Toru Yaita

Planted wire bit steering for logic and storage . . . . . 507 *W. F. Chow*
L. M. Spandorfer

A cryoelectronic distributed logic memory. . . . . 517 *B. A. Crane*
R. R. Laane

SCIENTIFIC PROGRAMMING APPLICATIONS
Trace-time-shared routines for analysis, classification and evaluation . . . . . 525 *Gerald H. Shure*
Robert J. Meeker
William H. Moore, Jr.

Degradation analysis of digitized signal transmission. . . . . 531 *J. C. Kim*
E. P. Kaiser

Digital systems for array radar. . . . . 541 *G. A. Champine*

TECHNIQUES IN PROGRAMMING LANGUAGES - PART II
DIAMAG: A multi-access system for on-line ALGOL programming . . 547 *A. Auroux*
J. Bellino
L. Bolliet

GRAF: Graphic additions to FORTRAN. . . . . 553 *A. Hurwitz*
J. P. Citron
J. B. Yeaton

The multilang on-line programming system . . . . . 559 *R. L. Wexelblat*

RPL: A data reduction language . . . . . 571 *Frank C. Bequaert*
Experimental automation information station: A1ST-O . . . . . 577 *A. P. Ershov, G. I. Kodzuhin, G. P. Makapov, M. I. Nechepurenko, I. V. Pottosin*

PAPERS OF SPECIAL INTEREST
Some issues of representation in a general problem solver . . . . . 583 *George W. Ernst, Allen Newell*
A compact data structure for storing, retrieving and manipulating line drawings . . . . . 601 *Andries Van Dam*
Experience using a time-shared multi-programming system with dynamic address relocation hardware . . . . . 611 *R. W. O'Neill*
THOR - A display based time sharing system . . . . . 623 *John McCarthy, Dan Brian, John Allen, Gary Feldman*

ADVANCES IN SOFTWARE DEVELOPMENT
"Compose/Produce: A user-oriented report generator for a time-shared system" . . . . . 635 *William D. Williams, Philip R. Bartram*
Code Generation for PIE (Parallel instruction execution) computers . . 641 *J. F. Thorlin*
Snuper Computer - A computer instrumentation automation . . . . . 645 *G. Estrin, D. Hopkins, B. Coggan, S. D. Crocker*

TECHNIQUES IN PROGRAMMING LANGUAGES - PART III
An algorithmic search procedure for program generation . . . . . 657 *M. H. Halstead, G. T. Uber, K. R. Gielow*
A system of macro-generation for ALGOL . . . . . 663 *H. Leroy*
COMMEN: A new approach to programming languages . . . . . 671 *Leo J. Cohen*
SPRINT: A direct approach to list-processing languages . . . . . 677 *Charles A. Kapps*
Syntax-checking and parsing of context - Free languages by pushdown-store automata . . . . . 685 *Victor B. Schneider*
The design and implementation of a table-driven compiler system . . . 691 *C. L. Liu, G. D. Chang*

SOME IDEAS FROM SWITCHING THEORY
Synthesis of switching functions, using complex logic modules . . . . . 699 *Y. N. Patt*
Adaptive systems of logic networks and binary memories . . . . . 707 *I. Alexander*
Design of diagnosable sequential machines . . . . . 713 *Zvi Kohavi, Pierre Lavallee*

NON-ROTATING MASS MEMORY
LCS (large core storage) utilization in theory and in practice . . . . . 719 *T. A. Humphrey*
Extended core storage for the control data 64/66000 systems . . . . . 729 *G. A. Jallen*
Simulation of an ECS-based operating system . . . . . 733 *M. H. MacDougall*

FAILURE FINDING IN LOGICAL SYSTEMS
A structural theory of machine diagnosis . . . . . 743 *C. V. Ramamoorthy*
Compiler level simulation of edge sensitive flip-flops . . . . . 757 *James T. Cain, Marlin H. Mickle, Lawrence P. McNamee*
A logic oriented diagnostic program . . . . . 761 *Herman Jacobowitz*
Automatic trouble isolation in duplex central controls employing matching . . . . . 765 *E. M. Prell*

COMPUTERS FOR INDUSTRIAL PROCESS ANALYSIS AND CONTROL
Digital backup for direct digital control . . . . . 771 *J. M. Lombardo*
Real-time monitoring of laboratory instruments . . . . . 777 *Paul A. C. Cook*
Industrial process control software with human attributes . . . . . 783 *J. B. Neblett, D. J. Brevik*

A resource-allocation scheme for multi-user on-line operation of a small computer

by ALLEN REITER

Lockheed Palo Alto Research Laboratory
Palo Alto, California

INTRODUCTION

Recently a great deal of interest has been evident in on-line information retrieval systems in which the retrieval search involves real-time man computer interaction. The advantages of such systems are becoming well known.[1] Operating systems designed to allow on-line time-shared information retrieval have common characteristics which distinguish them from other types of on-line systems. The system designer has to face problems in allocating the available computer resources that are not usually encountered in other contexts.

Information retrieval tasks typically involve many repeated accesses to data stored in not so rapidly accessible peripheral storage. When the system is being shared by several jobs simultaneously, the following situation results:

- The entire system is I/O bound.
- System performance times are critically affected by the order in which various I/O tasks are performed.
- It is obviously advantageous to have many different jobs in simultaneous execution, each placing requests into an I/O queue, so that the monitor can best schedule the order of peripheral storage accesses.
- It will not do to remove jobs from core temporarily by "rolling them out" into peripheral storage, since this will merely aggravate the I/O-boundedness of the system.
- Core storage has to be allocated dynamically, with care paid to both the present and the future needs of each job in execution.

This paper presents an approach to solving the system-design problems when the available computer resources, and in particular core storage, are quite limited. It is shown that it is indeed feasible to design a time-sharing information-retrieval system and obtain satisfactory working results on a small research laboratory sized computer not specifically intended for time-shared use.

This work was done in connection with a library reference-retrieval system designed at the Lockheed Palo Alto Research Laboratory. The system is currently running on the IBM System 360/30 computer, equipped with 32 thousand bytes of core storage, two disk-pack units, and one data-cell drive, all sharing the same I/O channel. A brief description of the retrieval system has appeared in print;[2] a later paper by D. L. Drew giving a full description of the time-sharing monitor will appear shortly.[3]

A generalized system

The system under discussion is designed to meet the following requirements, which are shared by most on-line multi-user information retrieval systems.

P1 (Multiple Simultaneous Use)—There are potentially several users on-line with the computer at any given moment in time, say 10, each of whom communicates with the computer via a slow telecommunication device. (The word *user* will be used synonymously with *console*, and will denote a source of human input to the computer.)

P2 (Independence)—Each user is completely independent of every other user. Nothing that transpires at one console shall have any influence on what occurs at any other console.

P3 (On-Line Interaction)—Each user's job involves many steps. By a *step* is meant user input followed by computer processing and system output. Any two consecutive steps of a job are necessarily separated

by the period of time it takes a user to initiate the new step; this period may last anywhere between several seconds to several minutes. The computer cannot be doing any processing for this job between two steps.

P4 (Program Segmentation)—A user's job requires the services of a group of program segments. These are selected by a monitor program from a large pool of such segments, based on an analysis of the user's input for each step. There is no a priori correlation between a given user and the set of program segments that he will require.

P5 (Storage Variability)—Different program segments place different core storage demands on the computer. These storage demands are of three types: storage for the string of segment code itself, storage for any working space that the segment might require, and storage for reading in data from the auxiliary memory devices.

P6 (Heavy File Manipulation)—Almost every step of a job will involve one or more accesses to data stored in auxiliary memory devices. This system requirement, together with its consequence that the operation will basically be I/0 bound, will motivate our approach to the system design.

P7 (Rapid Average Response Time)—The system should normally produce a response within a fixed time from the initiation of a step by a user. This fixed time may vary from step to step, depending on the nature of the task that the computer must perform within the step. Delays to a given user caused by other users should be guarded against; ideally, no user should even be aware that there are other users of the system at the same time.

In addition, without increasing the complexity of the system, we can add:

P8 (Priorities)—There are priority levels associated with each job. These priorities are not meant to be overriding, but are merely another factor to consider when the system monitor is allocating the system resources.

All these requirements are to be satisfied by a system which is to be implemented on a "small" computer. That is, the computer has a reasonably small core (high-speed) memory augmented by a large (but slow) random-access-memory device, and no built-in hardware for multiprocessing. Specifically, these limitations are:

RL1—There are too many program segements to fit into core simultaneously. The program segments will have to be stored in the random-access memory and fetched as needed.

RL2—There is not enough core storage to dedicate a storage area for each user. Core storage will be pooled and allocated to each user as needed. Each job is expected to return storage to the pool whenever it can do so.

RL3—The central processor of the computer can only be executing one program segment (application or monitor) at one time. The only activity that may go on in parallel with the processing is input/output.

As a consequence of P6 and RL1, it may safely be assumed that the system will be I/O bound to a large extent. In order not to increase the I/O load, the following specification is imposed on the system:

S1—Once a program segment has been brought into core and its execution has commenced, it must be allowed to come to its conclusion while it is still resident in core. Although the segment may relinquish control several times before it is finished, the segment is not to be "rolled out" to peripheral storage and brought back later in order to make its space available for other uses.

As will be seen, program segments will be relinquishing control quite frequently before being finished, and there may be several unrelated segments in core simultaneously. Taken in conjunction with RL2, this implies that the storage requirements of every program segment have to be scrutinized carefully before the program is loaded to make sure that the segment will be able to reach a successful conclusion and not get hung up because of lack of space.

The fact that I/O operations will have to be scheduled carefully implies:

S2—All I/O operations are to be performed by the system monitor. The individual program segments will place their respective requests into a queue which will be serviced by a system I/O scheduler.

A program segment can relinquish control to the system monitor (and thence to other program segments) for one or more of several reasons. First, it may have finished its task. Second, it may have just placed a request for I/O into the queue and cannot continue until the request has been acted upon. Third, it might need some additional core storage, and no core storage is currently available. Finally, its execution may be taking too long, and it interrupts itself to give the system a chance to do something else.

With a certain degree of arbitrariness it is being assumed that the system monitor will not be taking control away from the program segments, but rather the segments will relinquish control voluntarily. Thus:

S3.—The system is to be event driven, rather than time driven.

Because of the core storage limitations and the variation in the storage requirements from program segment to program segment, it is imperative to have some flexibility in handling storage. In particular,

if a contiguous block of storage sufficiently large to accommodate a storage request is not available, one should be able to honor the request utilizing non-contiguous areas. The editing problem in subsegmenting program and data areas is extremely complex, and in practice some compromise has to be effected between executing programs interpretively and placing restrictive demands upon the person writing the program segments. As this is not directly relevant to the allocation problem, we shall not go into this any further, but simply state:

S4—Storage requests can be honored by scatter-loading.

Resources to be allocated

The resources are to be allocated with the object of optimizing the performance of the total system. Priorities notwithstanding, overall system considerations will always override the considerations relative to a particular job. Thus, we do not intend to turn over control of all of the resources to jobs for fixed periods of time; the resource allocation algorithms will do their juggling on the basis of the *whole* picture at a given instant.

The system design is governed by two types of timing considerations: optimization with respect to total elapsed time, for all of the jobs currently on the computer, and optimization with respect to the average response time within a step, for each job and each console. No guarantee is made that every step along the line will receive a response within a certain allotted time span regardless of what is happening at the other consoles, not even if the particular job has the highest level; rather, it is hoped that long delays will be rare, and that the higher the priority the less frequent the delays. In fact, the *internal* priority of a given task for a given job will be a function of both its initial (external) priority and of the amount of time that has elapsed since the initiation of the step to which the task relates.

The first of the resources that will be allocated explicitly is that of core storage. When a job requests storage from the system, the monitor has to consider the following:

- Is enough free storage available to satisfy the request?
- Suppose that the request is granted, will the job make storage available to the rest of the system (because it is now able to run to a conclusion), thus freeing not only the area just granted, but other areas as well?
- Will the job be making additional requests for storage, and if so will the system be able to satisfy these requests?
- How will granting this request affect the storage requirements of the other program segments that are currently in core but are unfinished?

Of course, the monitor will expect each program segment to carry enough information about its own storage requirements to be able to answer all of these questions.

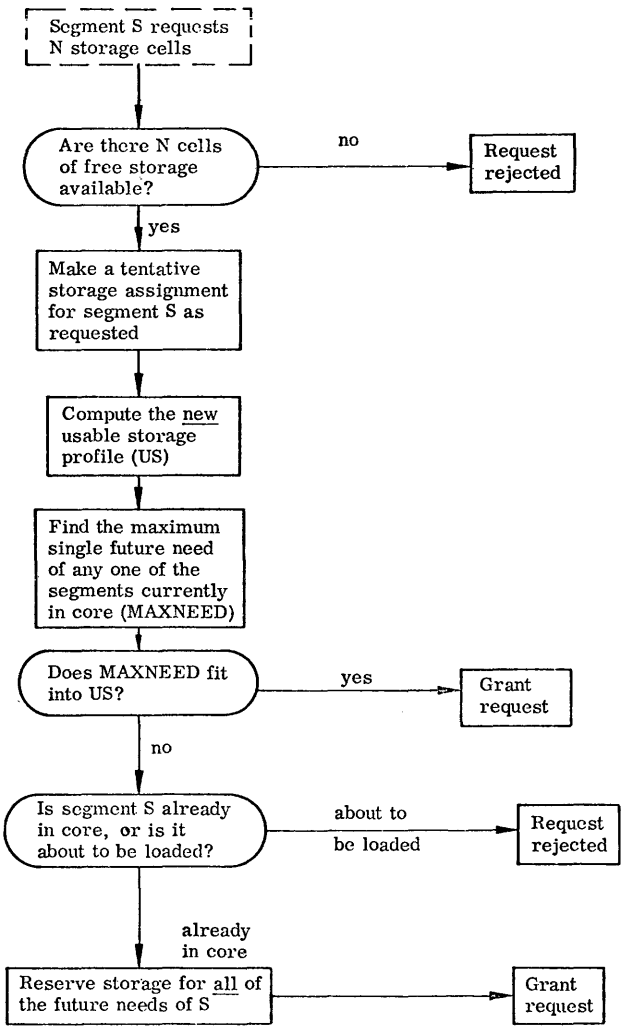


Figure 1—Processing storage requests

Another resource that has to be allocated carefully is that of access to the I/O devices. Although typically the hardware configuration is quite complex and varies from system to system, the only devices in the generalized system whose scheduling vitally affects the performance of the system are the random-access-memory (RAM) units. Without loss of generality it shall be assumed that only one RAM unit is on-line, and that its storage capacity is infinite. This device consists of cylinders, each of which is capable of containing many records. Associated with each I/O activity is a seek time, which we shall define as the time it

takes the RAM unit to respond to the I/O command before it can commence data transfer. (We are neglecting such things as latency times and read/write head selection times, which are small compared to seek times.) The seek time depends on the distance from the current cylinder to the one where the I/O is to take place, and is zero when the distance is zero.

To optimize the overall system performance, the monitor will try to schedule the order of operations on the RAM unit so as to minimize the total seek time. Although good data organization can effect great savings in the total seek time, the monitor will assume that records are to be accessed in a completely random order. If the users take the trouble to organize the data carefully they will be rewarded by faster response, but no such effort is demanded of them by the monitor.

The last resource that the system monitor will have to allocate among the several jobs is that of control of the central processor. Because of RL3, only one job can actually be running on the machine at any moment (apart from the I/O activity that may be going on simultaneously). The system monitor therefore has to decide (at various decision points in the course of operation) to which program segments control is to be transferred.

The core storage allocation problem

As already indicated, the processing of requests for storage is a complex affair. It must be remembered that as a result of P5 and RL2, each program segment which is in core or is about to be placed in core for execution has two kinds of storage requirements: (1) immediate storage for the string of code that makes up the body of the segment and for its immediate working storage and (2) future storage that the segment may yet require before it can come to an end but which it can currently do without.

If storage were to be reserved for each program segment at the time of its loading, there would be no particular allocation problem. When the monitor decided that the services of some given program segment were desired, it would try to load the segment. If enough storage were available at that time for all of the segment's needs, the segment would be loaded; otherwise, the loading would be postponed until a more propitious time. However, this mode of operation would mean that the system is not operating at the full capacity of the computer, and that much of the storage that might otherwise be used profitably by another job is being wasted.

Since we reject the "simple-minded" approach, we have to deal not only with the problem of fitting in the immediate needs of the segment at the time of its loading, but also with the problem of making certain that all of its later storage demands can be satisfied so that it can reach a conclusion and be removed from core. Moreover, the adoption of specification S1 prohibits us from satisfying these demands at the expense of some other program segment in core at that time.

In designing an allocation algorithm, care must be taken not to over-process. The elegance of any given scheme must be weighed against the storage requirements of the algorithm itself and against the time it takes the computer to execute the algorithm. It is possible to come up with an excellent resource allocation algorithm which unfortunately consumes most of these resources itself. A good algorithm should allow storage to be allocated efficiently and quickly in the common cases while at the same time making sure that when extreme cases arise they are taken care of (no matter how inefficiently). The algorithm outlined here is an attempt to proceed along these lines.

Several definitions are now in order. Any storage area that is unused (at the time under consideration) will be called *free*. If an area of core is at this time being used by a program segment, but that segment will be able to come to a conclusion without requiring any additional storage (thus freeing the area for the rest of the system), the area in question will be called *storage-in-waiting*, abbreviated SIW. Free storage and SIW together form *usable* storage. All storage which is not currently usable will be called *occupied.*

The procedure in determining whether there is sufficient room for a segment is to match the segment's immediate needs against the free storage, and its future needs against the left-over usable storage (which is guaranteed to become free at some later time). Segments will be loaded, even though at this moment in time they cannot run to a conclusion due to lack of space, as long as storage to satisfy their needs will become available eventually. Reserving storage is to be avoided whenever possible. The key to proper allocation is the division of each segment's needs into immediate and future, and taking the interplay of aggregate future needs into account.

This is accomplished as follows. When the system monitor receives a request for storage, it first checks the available free storage to see if the request can be accommodated. If not, the request is rejected (postponed) without further ado. Assuming that sufficient free storage is available, a tentative storage assignment satisfying the request is made. The storage needs of this particular segment are (tentatively) updated, and new SIW and usable storage profiles for the system are computed. Next, the monitor ob-

tains an estimate of the future needs of the entire system by computing the largest of the future needs of any single segment currently in core, including the segment about to be brought in (if the storage request is being made in order to bring in a new program segment). This largest future requirement is matched against the new usable storage. If the usable storage is sufficient to accommodate the future needs of every one (but not necessarily more than one) of the segments currently in core, the storage situation is adjudged to be favorable, the tentative storage assignment is made permanent, and the request is granted. This takes care of the great majority of the storage requests.

If, on the other hand, the maximum load is too big to be handled by the available usable storage, the storage situation is deemed unfavorable, and we have to consider the circumstances of the request. If the request is being made in order to bring in a new program segment, the request is rejected. New segments generally tend to increase the total load on the system; in fact, bringing in a new program segment is the only way that the load can be increased. Such rejection will follow any indication that the system may hang up if the new segment is admitted at this time, and not just those cases where hanging up is a certainty.

Assuming that the request has been made in order to accommodate the additional storage needs of some segment already in core, we know that its storage needs have been considered when it was first loaded, and that the request should be granted. It cannot, however, be granted without some precautions, since we took into account the needs of only one (the "hungriest" one) of the segments, and hence have to make sure that the monitor does not attempt to divide the available storage among several segments, ending with the insufficient storage for any of them. At this time a subset of usable storage is reserved for all of the future needs of the segment that is requesting storage. The free storage involved becomes part of the segment; any SIW storage involved will become part of the segment as soon as it becomes available. (See Figure 1.) All of the segment's future storage needs are thus taken care of, the segment will not be requesting any additional storage from the system, and hence by definition all of the area occupied by the segment becomes SIW. The net effect of granting the storage request on the system is one of increasing the usable storage. (This apparent paradox comes about as follows: If we were only concerned with maximizing usable storage, we would always reserve storage for a segment at the time of its loading. The effect of any reservation is a loss in the flexibility in the order of execution of various independent tasks, leading to a less efficient use of the other system resources. Thus, increasing usable storage can only be done at the cost of possibly not being able to later execute the segment that is "right" for that moment, a kind of "law of conservation of resources.")

The fact that storage is reserved for the segment means that the segment has to wait for it to become available, and cannot make use of any other storage that may in the meantime have become free. In very extreme cases it may happen that there are several segments waiting for the same area, and that we have set up a hierarchy as to the order of execution of these segments, regardless of other factors. This is not particularly disturbing, since the chain of circumstances leading to this situation will arise very rarely. It might be pointed out that even in the worst possible cases the system will be operating at least as efficiently as it would have been had we decided to reserve the necessary storage for every segment at the time of its loading.

An example will help illustrate the operation of the algorithm and the motivation behind the adopted conventions. Let us suppose that at some time t_0 program segments A and B are both in core. Program A occupies 1,000 cells, and will not require any more storage before it is done; B occupies 500 cells and will require an additional 1,500 cells; 1,000 cells are currently free. If B were to request the 1,500 cells at time t_0, the request would of course be rejected since the area is not available. However, we know that eventually A will be finished, and then B will be able to get its storage.

Let us suppose that it is desirable at this time to bring into core program segment C, which is currently residing in auxiliary memory. Segment C occupies 600 cells, and will need 2,000 more cells before it is done. Evidently, if we bring C in now (at time t_0), there will not be sufficient room left to complete either B or C (even after A is completed); hence, we postpone bringing in C until a more propitious time (i.e., until after we complete B).

If, on the other hand, at time t_0 segment D wants to come in, and D occupies 500 cells and will need another 1,000 eventually, then D is brought in. Even though there is no room at the present time to complete D, at some later time (t_1) segment A will be done, and then we can complete either B or D, depending on the order in which their requests come in.

We return to the case where C wants to come in, but this time assume that C only occupies 500 cells while still requiring an additional 2,000 later. It would seem that we should load C, since there would

still be enough room left to finish B (500 free locations plus the 1,000 cells currently belonging to A which are SIW), after which there would be room to complete C as well. Nevertheless, C is again refused admittance, for the following reason. Suppose that we do admit C at time t_0, and that at time t_1 C places a request for 500 cells. The 500 cells are free, but should they be granted to C then things would grind to a halt, as there would not be enough room left to complete either B or C. This would therefore mean that at t_0, when we are trying to load C, we also have to make a reservation of the remaining usable space for B. The situation becomes even messier if we have to contend with several segments like B in core at time t_0. To avoid overcomplicating the algorithm, we simply refuse to load C.

Note well the difference in loading D. If D is loaded at time t_0, and subsequently at time t_1 D places a request for 500 more cells, the request is granted and a reservation of the SIW space (currently held by A) is made for D; no reservations have to be made at time t_0, and no segments other than D have to be dealt with at time t_1.

It has been assumed in this discussion that the storage needs of segments can be satisfied by noncontiguous blocks of core. This assumption is not crucial, and the algorithm would work with minor modifications if the scatter-loading hypothesis S4 is dispensed with. The point is that now we have to worry not only about the total demands of a segment, but also about the "shape" in terms of the block sizes.*

The input / output scheduler

Unlike the core allocator, the I/O scheduler will not be intimately involved with individual program segments. Instead of being concerned with the aggregate of different segments that are resident in core at any moment in time, the I/O scheduler will be concerned with the aggregate of I/O requests in the queue, without regard (except for priority determination) for the segments that have placed the request. Thus, two distinct requests for I/O by the same segment will in no way be related to each other in the scheduling, and will be treated as if they were placed by different jobs; in particular, their order of execution is unpredictable.

The considerations involved in determining which I/O operation is to be scheduled next are:

(a) Which item in the queue would make for the lowest seek time?
(b) Which items have the highest urgency (in terms of the time that has elapsed since the initiation of the step leading to this request)?
(c) What are the external priorities of the jobs to which the items pertain?
(d) Which items are most favorable from the point of view of storage availability? (A request for input may or may not require obtaining the necessary storage from the system; a request for output may or may not imply that storage is being returned to the system.)

Having first eliminated from consideration all items (necessarily requests for input) for which the required space is not available, the scheduler will select from among the remaining items the best one, and this will be the request that will next be honored.

It shall be assumed that the data transfer rate of the RAM unit is very fast compared to the seek time. (This assumption is implicit in our failure to include the length of the I/O request in the list of factors to consider in the scheduling.) Since we have also assumed that the seek time in accessing items on the current cylinder is zero, it follows that any operation that can be performed without changing cylinders should be scheduled first, without regard for the other factors. If there is more than one operation that can be done on the same cylinder, other factors may be brought back into consideration; in practice it turns out that this does not significantly affect system performance, and that we might as well commence the I/O operation as soon as we find a request calling on the current cylinder, without further scanning of the queue.

If there are no items in the queue that call for the current cylinder, we have to associate a numerical weight with each item. This weight will be some function of factors (a) to (d). The item with the highest computed weight will be scheduled next.

Note that only one item at a time is being scheduled. This is consistent with the fact that both the contents of the queue and the storage situation may change significantly by the time the currently scheduledI/O operation is completed.

Also note that no explicit provisions are made to ensure that the entire queue will eventually be serviced. This is partly taken care of by including into consideration factor (b), and partly by the fact that I/O requests can be placed into the queue by program segments only when the latter have control of the central processor; hence reasonable care in sequencing the transfer of control from segment to segment will ensure that every item in the queue will eventually be acted on.

*For the Lockheed system, it is assumed that the program and data areas can be split into blocks of either 512, 1024, or 2048 bytes for noncontiguous loading; the block size is at the option of the programmer.

The control scheduler

The sequence in which the monitor passes control from segment to segment is not a particularly critical part of the overall system design. Since the system is by hypothesis I/O bound, there frequently will be times when *every* job is either waiting for access to the RAM unit or is waiting for user input, with the central processor being idle. The extent of the mutual interference between different users will ultimately be determined by how long each job has to wait for the I/O devices, rather than the availability of the central processor.

One possible way to proceed in trying to schedule the control transfer is to compute (at time t_0) for each job not currently in a "wait" state an internal priority function which will depend both on the external priority of the job and on the time that has elapsed since the job last had control (measured against some real time standard). Control would be given to the job with the highest internal priority. The internal priority function should rise sharply when the elapsed time passes some critical value, so that no job is left waiting too long regardless of priorities.

This scheme would guarantee that every job will eventually gain control, but will generally allow jobs with higher external priorities to gain control sooner.

CONCLUSION

The Lockheed LACONIQ system was designed along the lines indicated in this paper for the IBM System/360 family of machines. It is intended to be used almost exclusively in an information-retrieval setting. The major points in the system design and the motivation behind them are as follows:

(1) The operations for which this system is designed largely involve accesses to files stored in peripheral storage, with relatively little processing of the accessed information.

(2) Because records usually will be accessed in a random fashion, it is desirable that the system rather than the individual programmer schedule the required I/O operations.

(3) Because of 2, it is desirable to have as many jobs simultaneously in the computer as can be fitted into core in order to minimize the seek times of the random-access-memory devices.

(4) Because of 1, it is undesirable to dump a "core image" of a segment into peripheral storage for later retrieval in order to make core storage available for other jobs.

(5) Because of inherent core storage limitations, and because of the fact that the file manipulations will occasionally require that large blocks of core storage be made available for program segments, it is impractical to dedicate fixed areas of core storage for each user. Instead, core storage is pooled for the entire system and allocated out of this pool as required.

Points (2) and (5) distinguish LACONIQ from other on-line time-sharing systems designed for small computers that have been created to date, such as the MAC system[4] developed at MIT for the IBM 7094, and the DARTMOUTH system developed at Dartmouth College for a series of GE computers. The latter two were intended for general computation-oriented use and hence make liberal use of peripheral storage devices as an extension of core memory. It is the rather special orientation of LACONIQ that necessitates the delicate handling of the resource allocation problem.

Pending further experience with the system in actual and simulated environments, it is felt that satisfactory performance can be expected from the 360/30 with 32 thousand bytes of core storage if not more than four users are on the computer simultaneously. A 360/40 computer with similar core (whose performance will be tested in a simulation study) can be expected to accommodate 12 users at the same time. Increasing core storage should improve system performance slightly; it is felt, however, that system performance will level off rapidly and not be affected much by further increases in core storage. Since the system is I/O bound, the improvements in performance will be caused largely by allowing more I/O operations to take place simultaneously; i.e., by adding more input channels to the computer.

REFERENCES

1 J J ROCCIO G SALTON
AFIPS conference proceedings
Vol 27 part 1 293-305 Fall Joint Computer Conference 1965

2 D L DREW R K SUMMIT R I TANAKA R B WHITELEY
An on-line technical library reference retrieval system
American Documentation Jan 1966

3 D L DREW
The LACONIQ monitor: time sharing for on-line dialogues
to be published

4 A L SCHERR
Time-sharing measurement
Datamation 12 4 1966

Effects of scheduling on file memory operations*

by PETER J. DENNING
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

File system** activity is a prime factor affecting the throughput of any computing system, for the file memory is, in a very real sense, the heart of the system. No program can ever be executed without such secondary storage. It is here that input files are stored, that files resulting as the output of processes are written, that "scratchwork" required by operating processes may be placed. In a multiprogrammed computing system, there is a considerable burden on secondary storage, resulting both from constant traffic in and out of main memory (core memory), and from the large number of permanent files that may be in residence. It is clear that the demand placed on the file memory system is extraordinarily heavy; and it is essential that every part of the computing system interacting with file memory do so smoothly and efficiently.

In this paper we discuss some important aspects of file system organization relevant to maximizing the throughput, the average number of requests serviced per unit time. We investigate the proposition that the utilization of secondary memory systems can be significantly increased by scheduling requests so as to minimize mechanical access time. This proposition is investigated mathematically for secondary memory systems with fixed heads (such as drums) and for systems with moving heads (such as disks). In the case of fixed head drums the proposition is shown to be spectacularly true, with utilization increases by factors of 20 easily attainable. For moving head devices such as disks this proposition is still true, but not dramatically so; optimistic predictions look for at best 40 per cent improvement, an improvement attained only with serious risk that some requests receive no service at all. The solution lies at an unsuspected place: a method of "scanning" the arm back and forth across the disk, which behaves statistically like the access-time-minimizer but provides far better response than first come first served policies.

Other analyses of drum and disk systems, but from different viewpoints, have appeared elsewhere.[1,2,3] The importance of the results obtained here is that they are relatively independent of the behavior of the processes generating requests.

The model of the file system

For our purposes the computing system can be visualized as two basic entities, main memory and file memory. Only information residing in main memory can be processed. Since the cost of high-speed core memory is so high, there is at best limited availability of such memory space. A core allocation algorithm will be at work deciding which information is permitted to occupy main memory, and which must be returned to secondary memory; algorithms for doing so do not concern us here and are discussed elsewhere.[4] The essential point is that there may be considerable traffic between slow, secondary storage, and fast, core storage. System performance clearly depends on the efficiency with which these file memory operations take place. It is our concern here to discuss optimum scheduling policies for use by central file control.

We assume there is a basic unit of storage and transmission, called the *page*. Core memory is divided logically into blocks (pages); information is stored page by page on secondary storage devices; it follows that the unit of information transfer is the

*Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

**By "file memory" in this paper is meant secondary storage devices of fixed and movable head types. Drums are examples of fixed head devices, while both fixed and movable head disks exist. Tapes may be regarded as movable head devices.

page. We will assume that requests for file system use are for single pages, and that a process which desires to transfer several pages will have to generate several page requests, one for each page. Single-page requests are desirable for two reasons. First, read requests will be reading pages from the disk or drum, pages which are not guaranteed to be stored consecutively; it is highly undesirable to force a data channel to be idle during the inter-page delays that might occur, when other requests might have been serviced in the meantime. And, secondly, the file system hardware is designed to handle pages.

By far the biggest obstacle to efficient file system operation is that these devices, mechanical in nature, are not random-access. This means that each request must experience a mechanical positioning delay before the actual information transfer can begin. It is instructive to follow a single request through the file system. The various quantities of interest are displayed in Figure 1 and defined below.

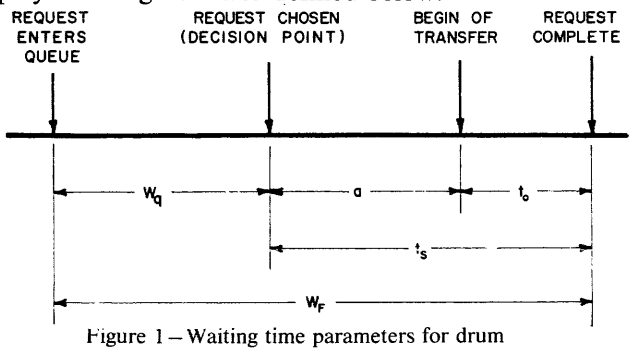


Figure 1 – Waiting time parameters for drum

t_o – the *transfer time*; the time required to read or write one page from or to file memory.

a – the *access time*; the mechanical positioning delay, measured from the moment the request is chosen from the queue to the moment the page transfer begins.

t_s – the *service time*; $t_s = a + t_o$

W_q – the *wait in queue*; measured from the moment the request enters the queue to the moment it is chosen for service.

W_F – the total time a request spends in the file system.

The policy

Before discussing scheduling policies, we should mention the goals a policy is to achieve. The policy we seek must maximize throughput, and yet not discriminate unjustly against any particular request. In addition, it must be reasonably inexpensive to implement. Since mechanical devices waste much time positioning themselves, a policy which explicitly attempts to minimize positioning time is suggested.

Because the file system does not operate rapidly enough to satisfy immediately the total demand, there is bound to be a queue. If a first-come-first-served (FCFS) policy is in effect, the queue is in reality nothing more than a by-product, a side-effect, of the system's inability to satisfy too heavy demands. It is natural to inquire whether the file system can take advantage of opportunities hidden within the queue. For example it might be possible to serve the last request in the queue during the positioning delay of the first request. Thus it is apparent that something may be gained by examination of the entire queue rather than just the first entry.

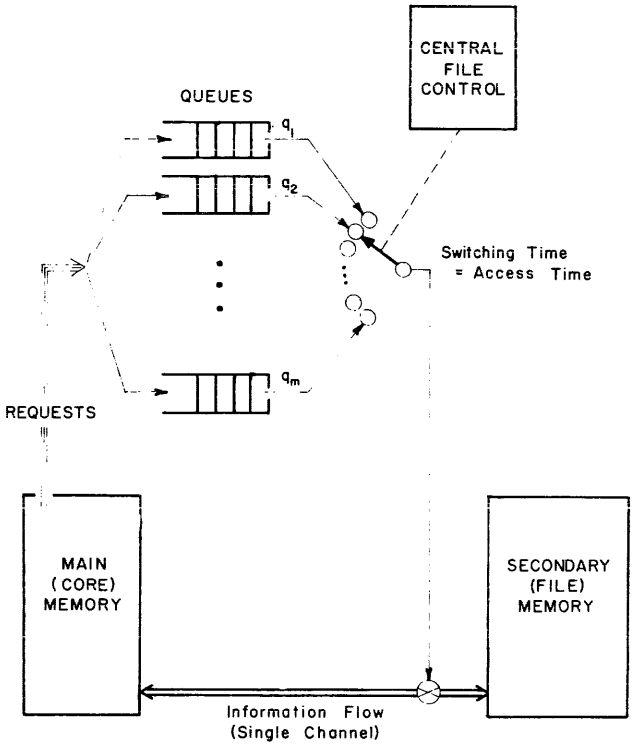


Figure 2 – Model of file system

In Figure 2 we see a generalized representation of the queueing activity for a file memory system. Incoming requests are sorted by starting address into one of the queues $q_1, q_2, \ldots, q_m$. We will see shortly that a drum is addressable by sectors, so in this case there is one queue for each sector, and two requests are in the same queue if and only if they are for the same sector. We will see that a disk is addressable by cylinders, so in this case there is one queue for each cylinder, and two requests are in the same queue if and only if they are for the same cylinder. The important point is this: each queue is associated with a mechanical position of the storage device. The switch is intended to illustrate the delays inherent in mechanical accessing.

If the device is a drum we may imagine that the switch arm is revolving at the same speed as the drum rotates, picking off the lead request in each queue

as it sweeps by. This structure is equivalent to the following policy: choose as next for service that request which requires the shortest access time; this policy will be called the shortest access time first (SATF) policy. It should be clear that the FCFS policy is less efficient because it involves longer access times per request. It should also be clear that SATF does not discriminate unjustly against any request.

Now suppose that secondary memory is a disk. Due to physical constraints, it is not possible to move the switch arm of figure 2 between q_1 and q_m without passing $q_2, \ldots, q_{m-1}$. The arm does not rotate, it may only sweep back and forth. However it can move arbitrarily from one queue to any other. The operation of moving the arm is known as a *seek*; but the policy shortest seek time first (SSTF), which corresponds to moving the switch arm the shortest possible distance, is unsatisfactory. For, as we will see, SSTF is likely to discriminate unreasonably against certain queues. In order to enforce good service we will find that the best policy is to scan the switch arm back and forth across the entire range from q_1 to q_m. We shall refer to this policy as SCAN. It will result in nearly the same disk efficiency as SSTF, but provide fair service. Again it should be clear that SCAN should be better than FCFS, which results in wasted arm motion.

The drum system

Our first goal is to analyze a fixed head storage device. The analysis deals with a drum system, but is sufficiently general for extension to other devices of similar structure. The analysis we use is approximate, intended only to obtain the expected waiting times, but it should illustrate clearly the advantages of scheduling.

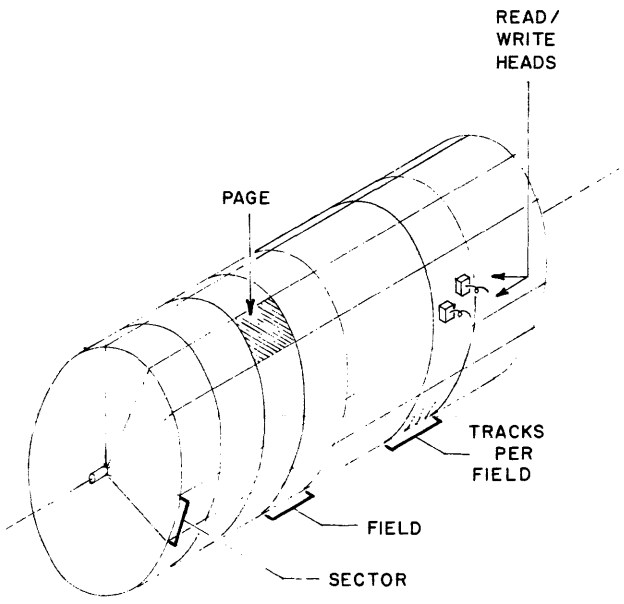


Figure 3 – Organization of the drum

The drum we consider is assumed to be organized as shown in Figure 3. We suppose that it is divided lengthwise into regions called *fields*; each field is a group of *tracks*; each track has its own *head* for reading and writing. The angular coordinate is divided into N *sectors*; the surface area enclosed in the intersection of a field and sector is a *page*. If the heads are currently being used for reading, they are said to be in *read status* (R-status); or if for writing, they are said to be in *write status* (W-status). Since there are two sets of amplifiers, one for reading, the other for writing, there is a *selection delay* involved in switching the status of the heads. Thus, suppose sector k is presently involved in a read (R) operation. If there is a request in the queue for a page on sector (k+1), and its operation is R, the request is serviced; but if the request is W, it cannot be satisfied until sector (k+2), since the write amplifiers cannot be switched in soon enough.

We suppose further that there is a drum allocation policy in operation which guarantees that there is at least one free page per sector. This might be done in the following way. A desired drum occupancy level (fraction of available pages used) is set as a parameter to the allocator. Whenever the current occupancy level exceeds the desired level, the allocator selects pages which have been unreferenced for the longest period of time and generates requests to move them to a lower level of storage (for instance a disk). Since these deletion requests are scheduled along with other requests, there will be some delay before the space is available, so the desired occupancy level must be set less than 100 per cent. A level of 90 per cent appears sufficient to insure that the overflow probability (the probability that a given sector has every field filled) will be quite small. If this is done, two assumptions follow: first, that a W-request may commence at once (if the heads are in W-status), or at most after a delay of one sector (if the heads are in R-status); and second, it is highly probable (though not necessarily true) that a sequence of W-requests generated by a single process will be stored consecutively. The assumption that W-request can begin at once is not unreasonable, and can be achieved at low cost.

We denote the probability that a request is an R-request by p and that it is a W-request by (1-p). That is,

$$\Pr[\text{a given request is R}] = p$$
$$\Pr[\text{a given request is W}] = 1 - p$$

We suppose that the access time per request is a random variable which can assume one of the N equally likely values

$$0, \frac{1}{N}T, \frac{2}{N}T, \ldots, \frac{N-1}{N}T$$

where T is the drum revolution time, N the number of sectors. When no attempt is made to minimize the access time, the expected access time is

$$E[a] = \sum_{k=0}^{N-1} \left(\frac{kT}{N}\right) \Pr[a = \frac{k}{N}] = \frac{N-1}{2N}T \quad (1)$$

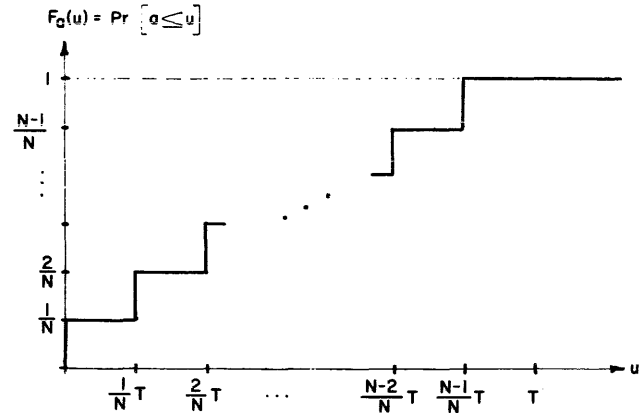


Figure 4 – Distribution of access-times $F_a(u)$

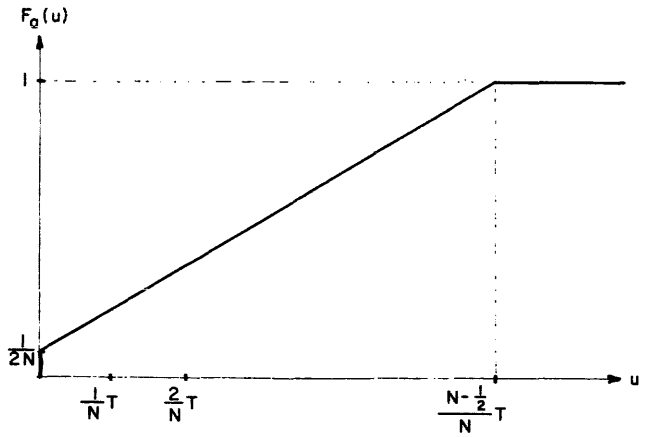


Figure 5 – Continuous approximation for $F_a(u)$

Note that E[a] is not exactly $\frac{T}{2}$. The distribution function $F_a(u) = \Pr[a \le u]$ is shown in Figures 4 and 5. Figure 4 is the actual distribution function, while Figure 5 is our approximation. This is used again in Appendix 2 for the formal derivation of the access time under the SATF policy.

There is some question whether this assumption of uniformity is valid; in fact it is not strictly so, for the following reason. In general a process will generate a sequence of single-page requests. If these are W-requests, and the drum allocation policy is functioning properly, there is a high probability these pages will be stored on consecutive sectors. It follows that a sizable subset of waiting R-requests, all originating from the same process, will want information which is stored on consecutive sectors, so that the access times will have higher probability of being small than a uniform distribution will assign. Although the use of a uniform distribution simplifies our calculations, it leads to conservative answers because the expectations obtained will tend to be too high.

If the service policy is FCFS the expected access time is just the access time for one request:

$$E[a] = \frac{N-1}{2N}T \quad (2)$$

Hence the *utilization factor* is

$$U_1 = \frac{\text{(page transfer time)}}{\text{(page transfer time)} + \text{(access time)}}$$

$$= \frac{\frac{T}{N}}{\frac{T}{N} + \frac{N-1}{N}\frac{T}{2}}$$

$$U_1 = \frac{2}{N+1} \quad (3)$$

where N is the number of sectors, and T the drum revolution time.

Rather than bore the reader with the detailed analysis of the SATF policy, we defer it to Appendix 2 and present immediately the result. The expected minimum access time when there are n requests in the queue is

$$E[a] = \left[\frac{T}{n+1}\left(1 - \frac{1}{2N}\right)^{n+1} + \frac{T(1-p)}{N}\right.$$

$$\left. + \frac{T(1-p)}{n+1}\left(1 - \frac{3}{2N}\right)^{n+1}\right]p^n$$

$$+ \frac{T}{N}p^{n+1}(1-p^n)\left[\left(\frac{1}{p} - \frac{1}{N}\right)^n - \left(1 - \frac{1}{N}\right)^n\right] \quad (4)$$

E[a] is to be interpreted as follows: if a request is selected at a time when it is one of n waiting requests, it can expect to experience a delay of E[a] seconds before the page transfer begins. Hence the utilization factor under the SATF policy is

$$U_2 = \frac{T/N}{E[a] + (T/N)} = \frac{T}{N\,E[a] + T} \quad (5)$$

where E[a] has been defined by equation (4). Note that the utilization factors U_1 and U_2 are not directly

dependent on the input distribution (that is, the interarrival time distribution of requests) and depend only on these parameters:

n – the number in the queue
N – the number of sectors
p – the probability a request is R.

We may define a *throughput factor* Q to be the number of requests serviced per unit time:

$$Q = \frac{1}{E[t_s]} \tag{6}$$

where $E[t_s]$ is the expected service time for the policy in force:

$$E[t_s] = E[a] + \frac{T}{N} \tag{7}$$

For the sake of obtaining estimates of W_F, the wait one request expects to experience in the file system, we will assume that the number n in the queues stays close to its expectation:

$$n \approx E[n]$$

It is well known[5] that in a FCFS queue, the expected wait is

$$W_F \approx E[n]\ E[t_s]$$

where $E[t_s]$ according to equation (7) is the expected service time. In our case

$$W_F \approx n\ E[t_s] \approx n\left(\frac{T}{2} + \frac{T}{N}\right) = nT\left(\frac{N+2}{2N}\right) \tag{8}$$

To obtain an estimate of W_F under the SATF policy, we reason as follows. If there are n requests waiting, then n/N of them are waiting for a given sector (see Figure 2). So when a request enters the file system, it expects to wait T/2 for its sector to come into position, plus n/N additional drum revolutions before receiving service, and finally T/N for its own page transfer. Hence

$$W_F = \frac{T}{2} + \frac{n}{N}T + \frac{T}{N} = T\left(\frac{1}{2} + \frac{n+1}{N}\right) \tag{9}$$

Example

To dramatize the effects of scheduling, we give a numerical example. A typical high-speed drum has a revolution time of 16.7 ms (3600 rpm). Typically there are 4096 words written around the drum's circumference. We use a page size is 64 words, yielding 64 sectors around the drum. Typically the probability of a read will be greater than that of a write, so the choice of 0.7 for p is not unreasonable. We have the following for the parameters:

T = 16.7 ms t_o = T/N = 0.26 ms
N = 64 sectors p = 0.7

Substituting these parameters into equations (2) to (9) yields the following table. We have chosen E[n] ≈n= 10.

POLICY	milliseconds E[a]	milliseconds $E[t_s]$	milliseconds W_F	Utilization U per cent	Throughput Q per second
FCFS	8.33	8.59	85.9	3	11.6
SATF	0.23	0.49	11.2	53	204.0

The relative improvement is

$$\frac{W_F\ [FCFS]}{W_F\ [SATF]} \approx \frac{nT\left(\frac{1}{2} + \frac{1}{N}\right)}{T\left(\frac{1}{2} + \frac{n+1}{N}\right)} = \frac{n(N+2)}{N + 2(n+1)}$$

For the given parameters this is 7.66, which means that incoming requests see a drum that is 766 per cent faster! Note that when n >>N, an overload condition, the improvement approaches a maximum of (N+2)/2.

Another calculation shows that U = 56 per cent for n = 20 under the SATF policy. It is apparent that with this rather simple policy the utilization of the drum is increased by a factor of almost 20! Another way to say this is that without scheduling, the drum is idle 97 per cent of the time, and with scheduling it is idle only 47 per cent of the time. It is apparent that scheduling for fixed head devices is well worth the cost, resulting in vastly increased throughput and utilization while providing fair service to requests.

As a final note we mention that some high-speed drums can switch head status so rapidly that the selection delay is negligible. In particular, a W-request may commence at once. The average behavior of the system under this assumption can be obtained as follows. First obtain the behavior of the system for read requests alone by setting p = 1 in equation (4) so that

$$E[a|\ \text{R-request}] = \frac{T}{n+1}\left(1 - \frac{1}{2N}\right)^{n+1} \tag{10}$$

Setting p = 1 in equation (4) cancels the effects of selection delay for W-requests used to derive E[a]. Since now

$$E[a\ |\text{W-request}] = 0 \tag{11}$$

if there is no selection delay, we obtain finally

$$E[a] = \frac{Tp}{n+1}\left(1 - \frac{1}{2N}\right)^{n+1} \tag{12}$$

The disk system

We turn our attention to movable head storage devices. Typical of equipment in this class is the movable arm disk, so we present an analysis for such a unit. There is sufficient generality that the results can be extended for use with other movable head devices of similar structure. First we will examine the behavior of a disk unit under access-time-minimizing scheduling policies and find that such policies do not provide comfortable increases in utilization. In fact they may give rise to very undesirable discrimination effects. Finally we will examine an alternative which leads to satisfactory disk behavior.

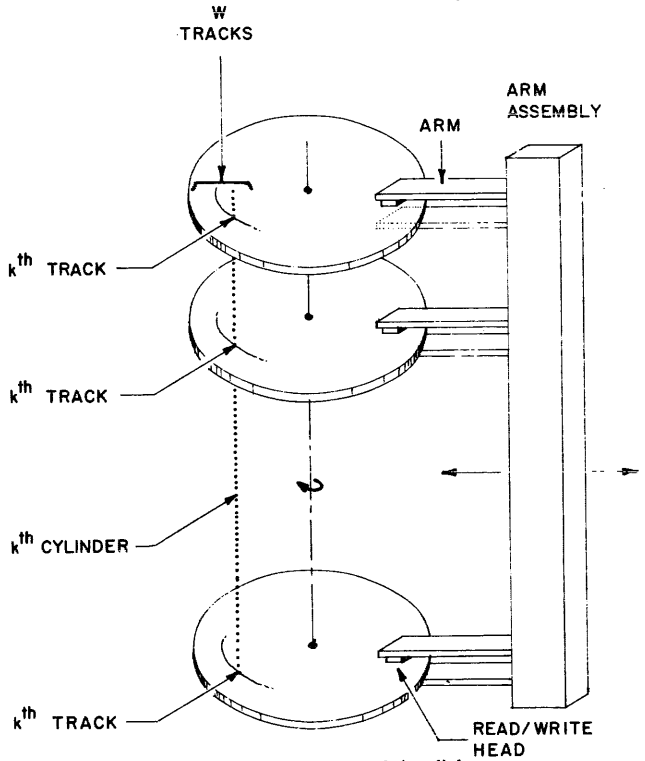


Figure 6 – Organization of the disk

The system we consider is displayed in Figure 6. There are a series of *disks* of like radii equally spaced along a common axis. Both surfaces of each disk are coated with a magnetic substance, so that information may be stored on both sides. The entire assembly rotates in time T. Each disk has a set of *tracks* situated near the outer edge, so that each track is of maximum length. One can imagine that the kth track of each disk is situated on the surface of a *cylinder*; thus, if the disk is W tracks wide, there are W concentric cylinders on which to store information. There is an assembly of movable *arms*, equipped with read/write heads. There is one set of heads for each side of each disk, sufficient to read or write one track, so each time a new track is requested, it is necessary to reposition the arms. The operation of positioning the arms is known as a *seek*. The mechansim for doing this is hydraulic, and is therefore inherently low. The *seek time* is illustrated in Figure

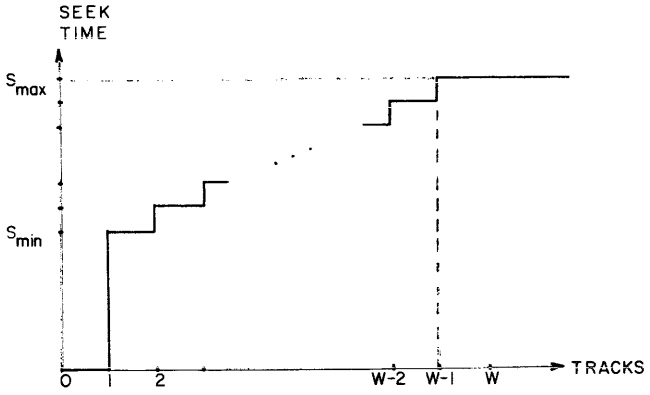


Figure 7 – Seek times

7, which indicates that a seek of one track requires a time S_{min}, while a seek the full width of the disk, (W-1) tracks, requires a time S_{max}. Typically S_{min} = 150 ms and S_{max} = 300 ms; thus, there is little difference between a seek of k tracks and a seek of (k+1) tracks. The bottleneck is getting the arms moving in the first place.

Once the arms have been positioned, any disk surface on the cylinder is addressable. We may regard the W concentric cylinders as a set of W drum systems by regarding each cylinder as a "drum". Unlike our drum model, we do not assume that there is always a free track on each cylinder on which to write a new page. This is because files stored on disks generally extend over several tracks, so it is futile to expect an availability of several tracks on the same cylinder. Therefore all requests are treated on an equal basis, without distinction between reads and writes.

Scheduling may be considered to operate on two levels. Waiting requests are sorted into W groups, one for each cylinder, in a manner indicated by Figure 2. The upper level of scheduling is to decide at which cylinder to position the arm. Once the arm is positioned, the lower level of scheduling comes into play. At this level, SATF scheduling could be used, but hardly seems worthwhile since the probability that more than one request is waiting for the same cylinder should, under normal load conditions, be quite small. So nothing is to be gained by scheduling policies other than FCFS within the cylinder queues. Thus the expected rotational delay is T/2. Our interest in disk scheduling will be confined to the level of deciding where to move the arms. In terms of pictures, we are interested in deciding how to move the rotary switch in Figure 2.

It is a common but questionable practice to chain tracks on the disk, the chaining information being

placed at the end of each track; the next track in the chain is therefore unknown until the current track has been read. We assume that if the End of File (end of a chain) has not been reached, that a new request is generated and scheduled on an equal basis with other requests; thus a file which unfortunately has become chained far and wide across the disk need not tie up the arm for more than one track at a time. In particular, a request requiring a shorter seek may be serviced in the interim between tracks of a given chain.

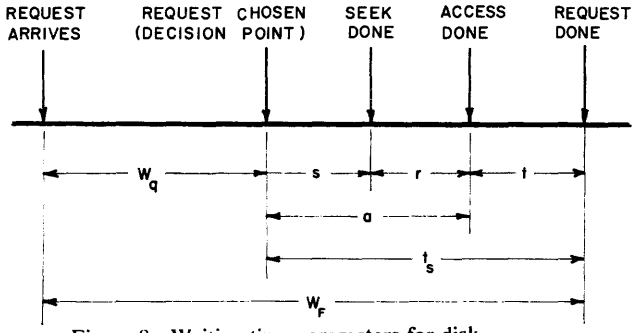


Figure 8—Waiting time parameters for disk

The disk waiting time parameters are shown in Figure 8. After a request enters the queue, it waits a time W_q until it is chosen for service. Once it has been chosen, a seek time s elapses. At the completion of the seek, an additional rotational delay r passes until the desired sector has come into position. Only then does transmission begin, and we assume it lasts for t seconds. The access time is

$$a = s + r \tag{13}$$

and the service time is

$$t_s = a + t = s + r + t \tag{14}$$

The waiting time in the system is

$$W_F = W_q + t_s \tag{15}$$

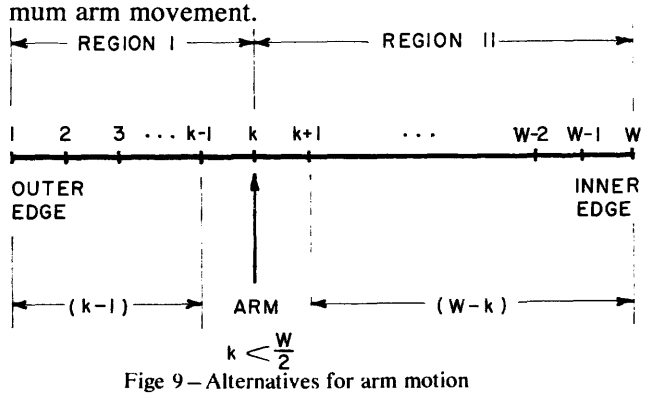
Our interest lies in deciding whether disk performance can be improved by planning the motion of the arm. Naturally we assume that, once the arms are positioned at a given cylinder, every request for that cylinder is served before another seek is initiated. The reason for this is apparent when one considers the magnitude of the minimum seek time, S_{min}, of Figure 7. Once every request for a given cylinder is satisfied, the arms are moved to a new cylinder; it would seem reasonable that a Shortest Seek Time First (SSTF) policy should be used. However this policy is unsatisfactory since it will tend to discriminate against requests for the inner and outer cylinders. To see that this is true, suppose the arm is at the kth track, where k<W/2, as indicated in Figure 9. Because requests are assumed to fall uniformly for any of the W tracks, it is apparent that there is more likelihood the arm will move toward the center of the track region; hence requests for the outer edges of the track region will tend not to be serviced. That is, since there are more requests in Region II than in Region I, the likelihood is greater that the arm moves into Region II.

Nevertheless it is entirely possible that the length of time a request for the extremities is delayed is still less than its normal wait under the FCFS policy. So before discarding the SSTF policy, we must analyze it to see whether the improvement in the utilization balances the undesirable effects. We will see that the improvement is marginal, that under heavy loading conditions undesirable discrimination effects occur. Our interest at this point is to analyze the SSTF policy; later we consider means to avoid anomalies.

Analysis of SSTF policy

As before, we do not present the detailed analysis of the SSTF policy here; instead we defer it to Appendix 3. We outline here the assumptions made. The analysis has as its primary goal to obtain approximate figures and expressions for average seeks and waits, so as to obtain utilization factors and characterize the effects of scheduling.

We treat the problem as follows. There are (at a decision point) n cylinders requested, and the arm is at cylinder k, where k is equally likely to be any cylinder. We ask: what is the expected seek time? The answer we will obtain is dependent on n, the number of requests in the queue, and on the disk parameters. To answer the question we find the distribution function for the arm movement caused by a single request; then we assume there are n identically distributed arm-movement random variables and use the result of Appendix 1 to find the expected minimum arm movement.



Fige 9—Alternatives for arm motion

In Figure 9 we show the arm positioned at track k. Suppose there is exactly one request waiting. How far will it cause the arm to move? To answer this we suppose that the arm does in fact move and later

multiply by the probability that it moves. If there are n cylinders requested, the probability that the present cylinder is *not* requested is

$$\left(1-\frac{1}{W}\right)^n \tag{16}$$

which is therefore just the probability a seek takes place. Now suppose the arm is positioned at cylinder k, where k<W/2. Given that the arm moves, we have

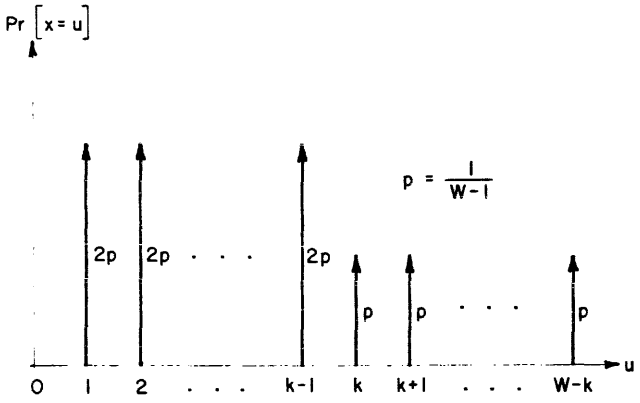


Figure 10—Conditional probability the arm moves u units

the probability distribution of Figure 10. The arms move 1, 2,..., (k−1) positions, each with probability 2p, or k, k+1,..., (W−k) positions, each with probability p. Here p is the probability a request is for a particular one of the remaining (W−1) cylinders:

$$p=\frac{1}{W-1} \tag{17}$$

To obtain a better understanding of Figure 10, it helps to imagine that Figure 9 has been folded on itself at position k. Since k<W/2, the first (k−1) positions of the folded distribution are twice as probable as the remaining positions. It is clear that the sitiation k>W/2 is identical, so we need consider only k<W/2, obtaining k>W/2 by symmetry.

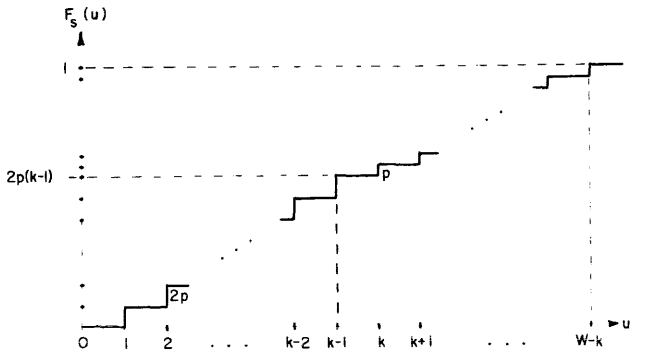


Figure 11—Cumulative distribution for arm motion $F_s(u)$

Figure 11 is the cumulative distribution $F_s(u)$ that the seek time s is less than or equal to u units:

$$F_s(u)=\Pr[s\leq u] \tag{18}$$

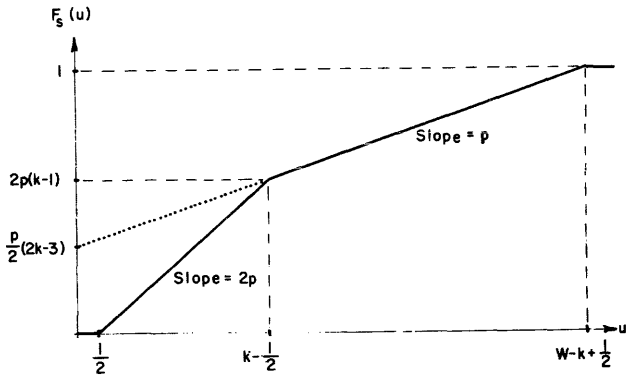


Figure 12—Continuous approximation for $F_s(u)$

Figure 12 is a continuous approximation to $F_s(u)$, obtained by passing straight lines through the centers of each plateau in Figure 11.

In Appendix 3 these assumptions are used to obtain the expected minimum seek time under the SSTF policy. It is

$$E[s]=\left(\frac{W-1}{W}\right)^n\left[\frac{1}{2}+S_{min}+\frac{S_{max}-S_{min}}{2(n+1)}\times\left(1+\frac{1}{n+2}\left(\frac{W}{W-1}\right)^{n+1}\right)\right] \tag{19}$$

The unscheduled (FCFS) seek time is obtained by setting n = 1 in equation (19) It is

$$E[s]=\frac{W-1}{W}\left[S_{min}+\frac{1}{2}+\frac{S_{max}\quad S_{min}}{4}\times\left(1+\frac{1}{3}\left(\frac{W}{W-1}\right)^2\right)\right] \tag{20}$$

Finally the access time is

$$E[a]=E[s]+\frac{T}{2} \tag{21}$$

Saturation and loading effects

We have said nothing about what SSTF is to do in case of a "tie"; that is, when the minimum arm movement is the same in either direction. Suppose we were to adopt the rule that, in case of a tie, the arm moves away from the center of the disk region; in this way (we would reason) we can counteract the effects of outer-edge discrimination. Consider what would happen if, on the average, there were one request per cylinder. There would almost always be a tie for a motion of one track, so that the arm could drift to one edge of the track region and stay there. Any requests for the opposite edge might be overlooked indefinitely.

This possibility that there may be times when there are many cylinder requests is the downfall of the SSTF policy. For under heavy loading conditions, the arm will tend to remain in one place on the disk.

and pay little or no attention to other regions. Although SSTF provides an increase in utilization, it is not at all clear that it will provide reasonable service to the requests. For these reasons we feel the following policy is superior.

The SCAN policy

As we mentioned in the discussion following Figure 2, a policy which insures reasonable service to requests is the SCAN policy, in which the switch arm is swept back and forth between q_1 and q_m, servicing any requests in the intervening queues as it passes by. We can think of operating the disk arm as a "shuttle bus" between the inner and outer edges of the track region, stopping the arm at any cylinder for which there are waiting requests.

Using average-value arguments we can obtain estimates for the utilization factor U and waiting time W_F of a single request. First assume that there are sufficiently many cylinders and sufficiently few requests that the probability of finding more than one request for a given cylinder is much less than the probability of finding just one request. This is equivalent to assuming that the total number of waiting request falls randomly within the track region, and that when it arrives the arm is at some random position within the track region. Then the expected distance between the arriving request and the arm is W/3. Now with probability 1/2 the arm is moving toward the request, in which case the distance the arm must travel before reaching the request is W/3. With probability 1/2 the arm is moving away from the request, in which case the distance the arm must travel before reaching the request is 3(W/3)=W. Hence the expected distance the arm moves before reaching the request is

$$\frac{1}{2}\left(\frac{W}{3}\right)+\frac{1}{2}W=\frac{2}{3}W \tag{22}$$

Next we obtain an estimate of the seek time per request. If there are n requests, these divide the track region in (n+1) regions of expected width W/(n+1). The seek time for one such distance is

$$E[s]=S_{min}+\frac{S_{max}-S_{min}}{n+1} \tag{23}$$

Assuming that, once the arm has stopped at a cylinder, there is an additional expected delay of T/2 for the starting address to rotate into position, we have for the access time

$$E[a]=E[s]+\frac{T}{2} \tag{24}$$

Now we can determine how long it takes the arm to cross the track region. It must make n stops and do a transfer of one track at each stop. So the time t between stops is

$$t=E[a]+T=E[s]+\frac{3}{2}T$$

or,

$$t=\frac{S_{max}-S_{min}}{n+1}+S_{min}+\frac{3}{2}T \tag{25}$$

Hence the time to cross W tracks is nt. The time W_F a single request must wait is, from equation (22),

$$W_F=\frac{2}{3}nt=\frac{2}{3}n\left[\frac{S_{max}-s_{min}+}{n+1}\ S_{min}\right]+nT \tag{26}$$

Example

To demonstrate the effects of scheduling, we present an example using the following parameters:

S_{min} = minimum seek time
= 150 ms
S_{max} = maximum seek time
= 300 ms
W = number of tracks
= 30
T = disk revolution time
= 60 ms
n = number of requests in queue
= 10

The utilization factor, or the fraction of time spent by the disk doing useful transmission, is

$$U=\frac{t}{E[a]+t} \tag{27}$$

where t is the transfer time for a single request. We use t=T, that is, each transfer is a full track. Due to uncertainties resulting from possible saturation effects we do not venture to give estimates of W_F under the SSTF policy. Using $W_F=n\,E[t_s]$ under the FCFS policy, $Q=1/E[t_s]$ for the throughput factor Q under all policies, and the disk equations just derived, we obtain the following table for the above parameters:

	milliseconds				Utilization U	Throughput Q
POLICY	E[s]	E[a]	$E[t_s]$	W_F	per cent	Per second
FCFS	195	225	285	2850	21.0	3.5
SSTF	113	143	203	?	29.5	4.9
SCAN	164	194	254	1700	23.6	3.9

It is apparent that with 10 cylinders requested the expected utilization increase with SSTF over FCFS is

$$\frac{29.5}{21.0}=1.40$$

so that the gain under SSTF is a not-too-impressive 40 per cent. It is the large minimum access time, S_{min}, that impairs efficiency. We conclude that, since the SSTF gain is marginal, the discrimination certain requests might suffer is not worth the risk. This substantiates our claim that SCAN is preferable. SCAN exhibits an improvement of

$$\frac{23.6}{21.0} = 1.12$$

or 12 per cent in utilization over FCFS; but the improvement in W_F,

$$\frac{2850}{1700} = 1.68$$

or 68 per cent is well worth the cost. It means that each request sees a disk that is 68 per cent faster. Finally note that the relative improvement

$$\frac{W_F\ [FCFS]}{W_F\ [SCAN]} \approx \frac{S_{min} + \frac{S_{max} - S_{min}}{3} + \frac{3}{2}T}{\frac{2}{3}\left(S_{min} + \frac{S_{max} - S_{min}}{n+1} + \frac{3}{2}T\right)}$$

approaches a maximum for

$$n >> \frac{S_{max} - S_{min}}{S_{min} + \frac{3}{2}T}$$

which is obtained under normal load conditions.

CONCLUSION

In the case of fixed head devices such as drums, we have seen that spectacular increases in utilization can be achieved by the rather simple expedient of Shortest Access Time First (SATF) scheduling. Requests serviced under the SATF policy are treated with the same degree of fairness as under the FCFS policy. Yet service is vastly improved.

We have seen, in the case of movable head devices such as disks, the impulse to use Shortest Seek Time First (SSTF) scheduling must be resisted. Not only may SSTF give rise to unreasonable discrimination against certain requests, but also during periods of heavy disk use it would result in no service to requests far from the current arm position. Enforcing good response to requests by shuttling the arm back and forth across the disk under the SCAN policy surmounts these difficulties, at the same time providing little more arm movement than SSTF. Hence SCAN is the best policy for use in disk scheduling.

W_F, the time one request expects to spend in the file system, provides a simple, dramatic measure of file system performance. Figures 13 and 14 indicate comparative trends in W_F for two policies. As usual, n is the number in the queue, T the device revolution time, N the number of sectors on the drum, S_{max} and S_{min} the maximum and minimum disk seek times respectively. Figure 13 shows that even for small n, W_F under the SATF policy is markedly improved. The curves diverge rapidly. We call attention to the asymptotic slopes, which apply under heavy load conditions. Figure 14 leads to similar conclusions for the SCAN policy. There can be no doubt that these simple policies—SATF for drum, SCAN for disk—can significantly improve file memory operations and thus the performance of the computing system as a whole.

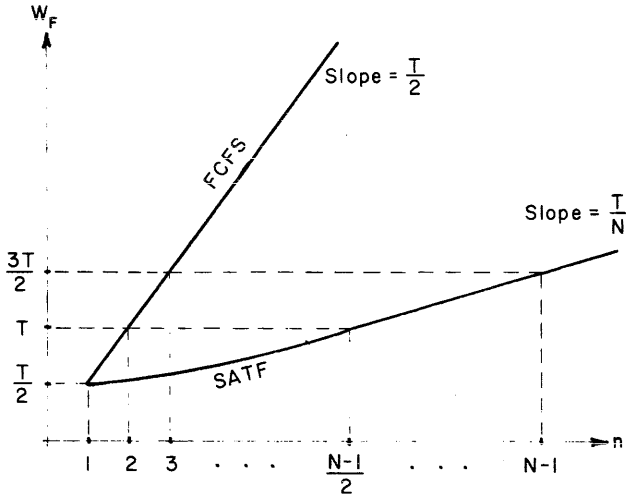


Figure 13—Comparison of W_F for two drum policies

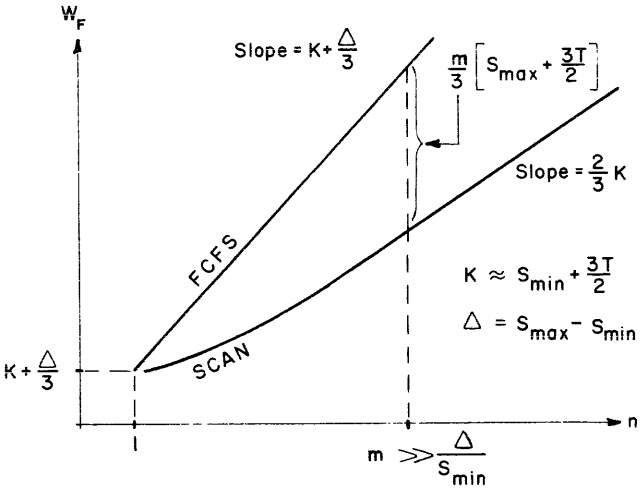


Figure 14—Comparison of W_F for two disk policies

ACKNOWLEDGMENT

The author wishes to thank Jack B. Dennis and Anatol W. Holt for helpful suggestions received while composing this paper.

Appendix 1—Expectation of the Minimum of Random Variables

Consider a set of independent, identically distributed random variables $t_1, t_2, \ldots, t_n$ each having density function p(t). Define a new random variable

(1.1) $\quad x = \min \{t_1, t_2, \ldots, t_n\}$

We wish to determine the expectation of x, E[x]. Now,

$$\begin{aligned}\Pr[x > \alpha[&= \Pr[t_1 > \alpha, t_2 > \alpha, \ldots, t_n > \alpha] \\ &= (\Pr[t > \alpha])^n \\ &= [F_t^c(\alpha)]^n\end{aligned}$$

where

$$F_t^c(\alpha) \equiv \Pr[t > \alpha] = \int_\alpha^\infty p(t)\,dt \tag{1.2}$$

is the complementary cumulative distribution for t. Using the fact that

$$E[x] = \int_0^\infty \Pr[x > u]\,du \tag{1.3}$$

we have

$$E[x] = \int_0^\infty (F_c^t(u))^n\,du \tag{1.4}$$

APPENDIX 2—Analysis of SATF Policy

Consider the SATF policy. As indicated by Figure 5, the distribution function for access time is

$$F_a(u) = \begin{cases} 0 & u \le 0 \\ \frac{1}{2N} + \frac{u}{T} & 0 < u \le \alpha \quad \alpha = \left(N - \frac{1}{2}\right)\frac{T}{N} \\ 1 & \alpha < u \end{cases} \tag{2.1}$$

where T is the drum revolution time, and N the number of sectors. Using Appendix 1,

$$E_1[a] = \int_0^\alpha \left(1 - \frac{1}{2N} - \frac{u}{T}\right)^n du = \frac{T}{n+1}\left(1 - \frac{1}{2N}\right)^{n+1} \tag{2.2}$$

here $E_1[a]$ is the expected access time if every request is R, the heads are in R-status, and n requests are in the queue. If the heads are in W-status they must be switched, and no transfer can begin for T/N seconds, until one sector has passed. That is, the distribution function when a change in head status must be made is

$$F_a(u) = \begin{cases} 0 & u \le \beta \quad \beta = \frac{T}{N} \\ \frac{1}{2N} + \frac{u}{T} & \beta < u \le \alpha \quad \alpha = \left(N - \frac{1}{2}\right)\frac{T}{N} \\ 1 & \alpha < u \end{cases} \tag{2.3}$$

From Appendix 1, the expected minimum access time is

$$\begin{aligned}E_2[a] &= \int_0^\beta du + \int_\beta^\alpha \left(1 - \frac{1}{2N} - \frac{u}{T}\right)^n du \\ &= \frac{T}{N} + \frac{T}{n+1}\left(1 - \frac{3}{2N}\right)^{n+1}\end{aligned} \tag{2.4}$$

If there is one or more W-request and the heads are already in W-status, the access time will be zero. If the heads are in R-status, *and* if none of the waiting R-requests is for the present sector, the access time will be one transfer time, T/N. Otherwise there is an R-request for the present sector, and so the access time is zero. Thus we must consider several cases. In the following, n is the number of requests in the queue, p is the probability that a given request is R, N is the number of sectors, and T is the drum revolution time.

Case 1—There are no W-requests in the queue, so that there are n R-requests. We must consider whether the heads are in R-status (with the same probability p that the last request was R) or in W-status (with probability 1-p).

Case 1a—The heads are in R-status. The distribution function of access times is given by equation (2.1), with expectation $E_1[a]$ given in equation (2.2). $E_1[a]$ occurs with probability p.

Case 1b—The heads are in W-status, so that at least one sector must pass before switching is completed. In this case no access of less than T/N can take place, so the distribution function is given by equation (2.3), with expectation $E_2[a]$ given by equation (2.4). $E_2[a]$ occurs with probability (1—p).

Case 1 summary—The probability that every request is R is p^n. Thus if every request is R we have

$$E_R[a] = E_1[a]\,p + E_2[a]\,(1-p)$$

and $E_R[a]$ occurs with probability p^n.

Case 2—There is at least one W-request, so that there are K = 0, 1, . . . ,(n—1) R-requests. We must consider two cases: the heads in W-status, and the heads in R-status.

Case 2a—The heads are in W-status, so that there need be no switching of amplifiers, and the transfer may begin at once. In this case E[a] = 0, with the probability that the last request was W. This is just (1—p).

Case 2b—The heads are in R-status. The access time then will be zero if there is an R-request for the present sector and T/N otherwise. This situation occurs with probability p. Now, for each k = 0, 1, . . . ,(n—1) the probability that there are k R-requests and (n—k) W-requests is

$\binom{n}{k} p^k (1-p)^{n-k}$ where $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$

The probability that none of these is for the present sector is

$$\left(1 - \frac{1}{N}\right)^k$$

That is, the access time is non-zero only if all k R-requests are not for the current sector. Hence the probability that there are k R-requests and none of them is for the present sector is

$$\left(1 - \frac{1}{N}\right)^k \binom{n}{k} p^k (1-p)^k$$

Since this is true for each k = 0, 1, . . . ,(n−1), we must sum over k.

$$\binom{n}{k}\left(1 - \frac{1}{N}\right)^k p^k (1-p)^{n-k}$$

$$= \binom{n}{k}\left[p\left(1 - \frac{1}{N}\right)\right]^k \left(1-p\right)^{n-k} - p^n\left(1 - \frac{1}{N}\right)^n$$

$$= \left[1 - p + p\left(1 - \frac{1}{N}\right)\right]^n - p^n\left(1 - \frac{1}{N}\right)^n$$

$$= \left(1 - \frac{p}{N}\right)^n - p^n\left(1 - \frac{1}{N}\right)^n$$

Hence the probability

$$Q = \Pr\left\{\begin{array}{l}\text{no R-request for the present sector } \textit{and} \text{ heads}\\ \text{in R-status } \textit{and} \text{ at least one W-request}\end{array}\right\}$$

$$Q = p^{n+1}\left[\left(\frac{1}{p} - \frac{1}{N}\right)^n - \left(1 - \frac{1}{N}\right)^n\right]\left(1 - p^n\right)$$

(2.6)

But Q is just the probability that $E[a] = \frac{T}{N}$. Collecting together the results of cases 1 and 2 we see that

$$E[a] = E_R[a]\, p^n + \frac{T}{N} Q$$

(2.7)

so that

$$E[a] = \left[\frac{Tp}{n+1}\left(1 - \frac{1}{2N}\right)^{n+1} + \frac{T(1-p)}{N} + \frac{T(1-p)^{n+1}}{n+1}\right] p^n$$

(2.8)

$$\left(1 - \frac{3}{2N}\right)^{n+1}\Bigg] p^n + \frac{T}{N} p^{n+1}\left(1 - p^n\right)$$

$$\left[\left(\frac{1}{p} - \frac{1}{N}\right)^n - \left(1 - \frac{1}{N}\right)^n\right]$$

which concludes the derivation.

APPENDIX 3 – Analysis of SSTF Policy

From Figure 11 we have for the distribution function for the arm movement for one request, conditioned on the arm moving:

$$F_s(u) = \begin{cases} 0 & u \leq \frac{1}{2} \\ 2pu - p & \frac{1}{2} < u \leq (k - \frac{1}{2}) \\ p + \frac{p}{2}(2k-3) & (k - \frac{1}{2}) < u \leq (W - k + \frac{1}{2}) \\ 1 & (W - k + \frac{1}{2}) < u \end{cases}$$

This is true for k < W/2. By symmetry, the same is true for k > W/2, with k replaced by (W−k). From Appendix 1, the expected minimum seek time is

$$E[s] = \int_0^\infty \left(1 - F_s(u)\right)^n du$$

(3.1)

$$= \int_0^{\frac{1}{2}} du + \int_{\frac{1}{2}}^{k-\frac{1}{2}} \left(1 + p - 2pu\right)^n du + \int_{k-\frac{1}{2}}^{W-k+\frac{1}{2}} \left(1 - \frac{p}{2}(2k-3) - pu\right)^n du$$

(3.2)

This integration is straightforward, so we do not reproduce it here. The result is

$$E[s] = \frac{1}{2} + \frac{1}{2p}\left[1 - \left(1 - 2p(k-1)\right)^{n+1}\right]\frac{1}{n+1} + \frac{1}{p(n+1)}\left(1 - 2p(k-1)\right)^{n+1}$$

(3.3)

Equation (3.3) applies for k < W/2. By symmetry (Figure 9) it applies for k > W/2 if we replace k by (W−k). The case k < W/2 occurs with probability 1/2, and so does the case k > W/2. Hence, collecting terms in equation (3.3),

$$E[s \mid k < \frac{W}{2}] = \frac{1}{2} + \frac{1}{2p(n+1)}\left[1 + \left(1 - 2p(k-1)\right)^{n+1}\right]$$

(3.4)

By symmetry, $E[s \mid k > \frac{W}{2}]$ is the same. Then

$$E[s] = \frac{1}{2} E[s \mid k < \frac{W}{2}] + \frac{1}{2} E[s \mid k > \frac{W}{2}]$$

$$E[s] = E[s \mid k < \frac{W}{2}]$$

(3.5)

Now, recalling that the head position k is assumed to be uniform for values of k = 1, 2, . . . ,W, we can approximate its distribution function by a ramp of

slope 1/W from $\frac{1}{2}$ to $W+\frac{1}{2}$. Conditioning on $k < \frac{W}{2}$, we can approximate the conditional distribution by a ramp of slope 2/W from $\frac{1}{2}$ to $\frac{1}{2}\left(W+1\right)$. Therefore we integrate $\left(\frac{2}{W} E[s \mid k < \frac{W}{2}]\right)$ on the interval $\left[\frac{1}{2}, \frac{1}{2}\left(W+1\right)\right]$. The result is, using $p = 1/(W-1)$:

$$E[s] = \frac{1}{2} + \frac{W-1}{2(n+1)}\left[1 + \frac{1}{n+2}\left(\frac{W}{W-1}\right)^{n+1}\right] \quad (3.6)$$

recalling that this is conditioned on the arm moving, we remove this condition by multiplying by the probability the arm moves:

$$\left(1 - \frac{1}{W}\right)^n = \left(\frac{W-1}{W}\right)^n$$

and obtain the following:

$$E[s] = \left(\frac{W-1}{W}\right)^n \left[\frac{1}{2} + \frac{W-1}{2(n+1)}\left(1 + \frac{1}{n+2}\left(\frac{W}{W-1}\right)^{n+1}\right)\right] \quad (3.7)$$

Noting that the time required for (W−1) tracks is S_{max}, and the time for one track is S_{min} (if the arm moves), we have for the expected minimum seek time when n are in the queue:

$$E[s] = \left(\frac{W-1}{W}\right)^n \left[\frac{1}{2} + S_{min} + \frac{S_{max} - S_{min}}{2(n+1)}\right.$$
$$\left.\left(1 + \frac{1}{n+2}\left(\frac{W}{W-1}\right)^{n+1}\right)\right] \quad (3.8)$$

which concludes the derivation.

REFERENCES

1 A WEINGARTEN
The Eschenback drum scheme
Comm ACM Vol 5 No 7 July 1966

2 D W FIFE J L SMITH
Transmission capacity of disk storage systems with concurrent arm positioning
IEEE Transactions on Electronic Computers Vol EC-14 No 4 August 1965

3 P J DENNING
Queueing models for file memory operations
MIT Project MAC Technical Report MAC-TR-21 October 1965

4 L A BELADY
A study of replacement algorithms for a virtual-storage computer
IBM Systems Journal Vol 5 No 2 1966

5 T L SAATY
Elements of queueing theory
McGraw-Hill Book Co Inc pp 40 ff New York 1961

Address mapping and the control of access in an interactive computer*

by DAVID C. EVANS
University of Utah
Salt Lake City, Utah**

and

JEAN YVES LECLERC
109 rue St. Dominique
Paris, France

*This work was sponsored by the Advanced Research Projects Agency, Department of Defense, under contract SD-185.
**Formerly of the University of California, Berkeley

INTRODUCTION

Computer system designs have attempted to maximize machine utilization (sometimes called efficiency) without regard to other important factors including the human effort and elapsed time required to solve problems. In recent years some computing systems have been developed which provide communication and control means by which users can interact with their own computing processes and with each other. Initial use of these systems has demonstrated their superiority in computer-aided problem solving applications. These interactive computing systems are mainly adaptations of conventional computing systems and are far from ideal in many respects. This paper describes a much improved mechanism for protection, address mapping, and subroutine linkage for an interactive computing system.

The user of a computing system should be able to interact arbitrarily with the system, his own computing processes, and other users in a controlled manner. He should have access to a large information storage and retrieval system usually called the file system. The file system should allow access by all users to information in a way which permits selectively controlled privacy and security of information. A user should be able to partition his computation into semi-independent tasks having controlled communication and interaction among the tasks. Such capability should reduce the human effort required to construct, debug, and modify programs and should make possible increased reliability of programs. The system should not arbitrarily limit the use of input/output equipment or limit input/output programming by the user.

The system capability specified above is available to some degree in present computing systems. The particular limitations to which this paper is directed are the following:

(1) The familiar memory protection systems limit or control access to specified regions of physical memory or to specified units of information. The actual system need is to control the various access paths to information. For example, at one time a particular segment of information may be a procedure for execution only to itself, a data segment to a debugging program, and entirely inaccessible to the other segments in memory.
(2) In order to protect the input/output equipment from unauthorized use, most multiple access computing systems deny all direct access to input/output equipment by user programs. The system described permits selective controlled access without limit as authorized by the executive system. For example, a user may be given unrestricted use of a magnetic tape unit without hazard to other users or to the system.
(3) Most systems require modification of procedures by program to bind segments together for a computing process. This is usually done by means of a "loading routine." The system de-

scribed permits arbitrary binding at run time with no modification of data or procedure and no compromise in protection of information.

(4) It is often the case that a computing task is made up of a number of semi-independent computing processes which should operate concurrently with only limited interaction. Familiar computing systems do not provide a convenient means for handling such cases.

Before examining the mechanisms proposed let us examine a computing process. We may think of a process as an activation of a procedure which operates on other elements of information. We call these elements parameters and the procedure itself the base parameter of the process. Each parameter may be a simple element, such as a data element, or may be a compound element, such as another procedure with its parameters. Ordinarily, elements of a process as described are segments of information from the file system. Segments either may be private to a single process or user or may be shared among users or processes.

The subject of this paper is a mapping mechanism by which procedures are bound to their parameters at execution time without modification or relocation. Three functions are performed by means of this single mechanism. (1) It permits each procedure to see its parameters through a map which defines the relationship between its own address space and the address spaces of its parameters. (2) It permits the user (or the process) to selectively control access to information on every access path. (3) It permits selective control by the user and the system of access to processing resources, including input/output equipment and instructions.

Several address mapping schemes have been described previously which do not provide all of the desired capability.[1,2,3,4,5] The most serious defect of such schemes is that access to a segment of information solely depends on that segment of information, but as will be justified later, should depend on the access path from the parameter making the access to the parameter being accessed.

Computing processes and addressing spaces

Before examining computing processes and their control, we will make the following definitions:

Segment

Information is stored in a file system. The basic entity of the file system is a segment which is an ordered collection of information having a symbolic name as in the terminology of J. Dennis.[4] Examples of such segments might be the body of a procedure, an array, a pushdown stack, a free storage list, or a string storage block. A segment may be a portion of another segment or may be a set of segments each of which also has a symbolic name.

Parameters and parameter spaces

The relation between two segments depends on those two segments and not only on the called segment. For example, consider the debugging program DDT and the program P:

P may not write on P,
DDT may write on P,
DDT may not write on DDT, and so on.

The user should be given tools which permit him to establish the proper relations between segments.

The structure of a computing process may be represented as a tree. Consider the structure represented in Figure 1. In this structure the procedure segment named P1 may directly access the procedure segment named P2 and data segments named S1 and S2. The procedure segment P2 may directly access the data segment S1. In the example shown it is not necessary for P1 to refer to S1 by the same addresses as are used by P2 to refer to S1.

We have chosen the following more convenient notation which is an adaptation of a functional notation due to Von Neumann to represent information of the type shown in Figure 1. If the computing process of the figure is called α, it is represented in the notation as

$$\alpha\text{: (P1, (P2, S1), S1, S2)}$$

This may also be represented in a less condensed way:

$$\alpha\text{: (P1, } \beta\text{, S1, S2)}$$
$$\beta\text{: (P2, S1)}$$

The three representatives are equivalent.

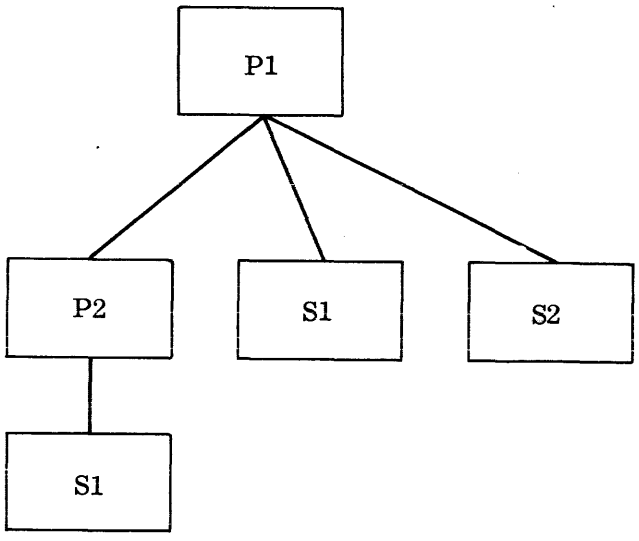


Figure 1 – The representation of the access paths of a computing process

Consider now the procedure segment P1. It may be represented as

(P1, F1, F2, F3)

where P1 is called the root parameter and F1, F2, and F3 are formal parameters of the procedure. If the procedure P1 is to be executed, it is necessary that actual parameters be assigned in place of the formal parameters F1, F2, and F3 as has been done in the expression,

(P1, β, S1, S2)

It may be seen that an actual parameter may be a simple segment as is S1, or it may be a compound structure as is β.

Each segment (either a procedure or data segment) may refer to other segments. These segments are the "parameters" of the given segment which is called the root segment. Such a segment SO with its associated actual parameters S1, S2, S3, ... SN is written as before,

(S0, S1, S2,... SN)

and defines a "parameter space."* Two parameter spaces having the same root segment are different if any parameter is different. The root segment of a parameter space is always fixed and sees itself as parameter number 0. It references other parameters with numbers 1, 2, 3,... N. Every address appearing in the memory of the computer (e.g., in instructions) will refer to such a parameter space in the form

PARAMETER, DISPLACEMENT

where DISPLACEMENT identifies an element of a parameter. The parameter space to which an address is relative will be called the "source" of that address.

Note that a parameter space may be a parameter of a parameter space. In the example given β is a parameter space which is a parameter of the parameter space α.

There are several kinds of parameters: A parameter may be fixed for that parameter space or may be a dummy parameter which is changed every time the parameter space is entered. A parameter may be a scratch segment in which case every time the parameter space is entered, the system assigns a fresh copy of the scratch segment for temporary storage, which is released when the program returns from the parameter space. Such a parameter should be declared, but the user can rely on the system for getting fresh copies. This can save space in the secondary memory, because a pool of scratch segments can be shared by the system among the many users of such segments.

User space

Each parameter space which has been defined is given a number for identification. The number is assigned by the system and has no meaning to the users. Each segment which is not the root segment of a parameter space is considered to be a parameter space and is also given a number. These parameter space numbers are the addresses of the parameter spaces in "user space." It is necessary for the system to know the identity of the parameter space to which an address is relevant. An address in user space has three components.

SOURCE, PARAMETER, DISPI ACEMENT.

An address in user space is an unambiguous address specification which will be used explicitly in subroutine linkages. Users may freely manipulate parameter space addresses without hazard to the system or ot other users in the system described in this paper. Operations on user space addresses must be limited by the system to assure the system integrity and to protect the computing processes of users from accidental or intentional damage by other users.

1. A process may request and receive the information which specifies the status of an immediate dependent.
2. A process may transmit an interrupt signal up the hierarchy which will be received and acted upon by the first superior process in which the corresponding interrupt is armed. If no process in the user's hierarchy of processes is armed, the interrupt will be acted upon by the system itself.
3. Processor-generated interrupt signals indicating faults, attempted violations of protection or other information may be transmitted up the hierarchy as in No. 2 above.

It is evident from the above that there are two means for defining communications among processes. Each process should be armed for the desired interrupt signals. The desired common data segments should be assigned so that processes may obtain messages from other processes. A process may have the capability to alter its priority or its interrupt arming.

Dynamic binding

Binding is the assignment of actual parameters to a parameter space. In traditional computing systems the only space which exists at execution time is the physical address space. Binding is done in the physical address space before execution by a program called loader, assembler, or compiler which modifies the

*The term parameter space includes the term subprocess and parameter space reference 6.

stored addresses. In the system described, the mapping hardware binds parameter spaces at execution time with no modification of the addresses stored in memory and without the sacrifice of protection that results when conventional base registers are used for the binding.

When a parameter space is entered it is not always completely defined. The definition is complete when all formal parameters in the parameter space have been replaced by actual parameters. An actual parameter may, of course, be a value, a data segment, a procedure segment, a parameter space, or some descriptor of the parameter. This may be accomplished by the transmission of some information from the calling parameter space to the called parameter space as in the transmission of arguments to a subroutine, or it may be accomplished by the execution itself.

The mapping functions

The address of any item is uniquely defined in user space by the three components

SOURCE, PARAMETER, DISPLACEMENT

During execution these three components are always known although only the parameter and displacement components are stored in the text of the procedure. The mapping function translates addresses from user space to physical address space. The mapping function may be implemented in any of several forms. Only one implementation will be described in this paper. Alternative implementations are described in reference 6, and their relative merits are examined.

In the implementation described, it is assumed that a conventional pageing strategy as described in reference 1 is used for memory allocation. In this scheme the physical memory is partitioned into a set of uniform sized blocks. The physical address may thus be represented by two components,

BLOCK, LINE

where BLOCK is the number of the physical partition and LINE is the number of the element of the block. Block size is determined by the same criteria as in other paged memory systems.

An associative implementation of the address translation map is shown in Figure 2. The address in user space is stored in registers marked SOURCE 0, PARAMETER, DISPLACEMENT. The PARAMETER, DISPLACEMENT portion is contained in the address field of an instruction as stored in memory or an instruction address register. The DISPLACEMENT field has been partitioned into subfields called

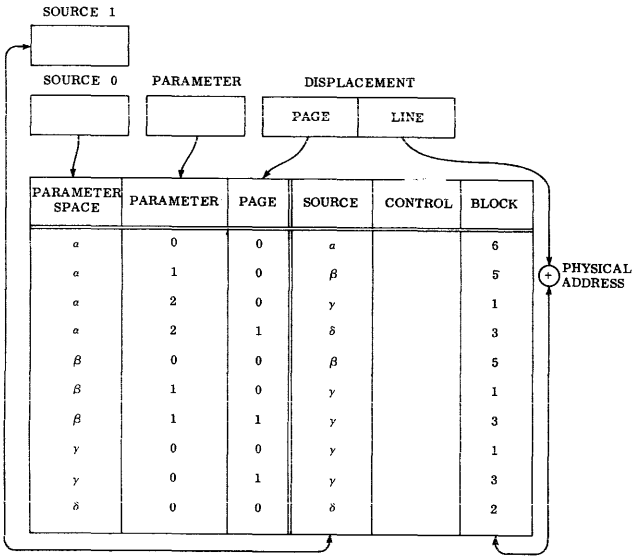


Figure 2 – Associative implementation of address translation map

PAGE and LINE. A page of information is stored in memory in a block. In order to understand the operation of the map consider the address structure used as an example in part B. There are four parameter spaces:

α: (P1, β, S1, S2)
β: (P2, S1)
γ: S1
δ: S2

Consider that the information is stored in the blocks of physical memory as indicated in Figure 3.

The segment P1 is one page in size and is stored in block 6. The segment S1 requires two blocks for storage; page 0 is stored in block 1, and page 1 is stored in block 3. Other segments are stored as indicated.

This storage allocation and address structure is represented in the address transformation map as shown in Figure 2. The first line of the map represents the parameter P1 of the parameter space (source) α. Any address retrieved from P1 when accessed through this table entry is relative to the parameter space α as indicated in column 4 of the first row. The information

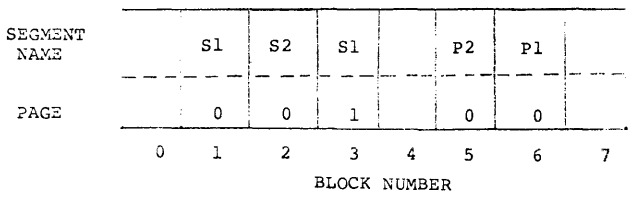


Figure 3 – Representation of storage assignment in physical memory

is stored in block 6 of memory as indicated in column 6. The second line represents parameter 1 of parameter space α, which is parameter space β. The only part of the parameter space β which is directly addressable from α is P2, the root parameter of β, which is a one page segment stored in block 5. Each page of every parameter space is similarly represented by a row in the table.

Consider now the use of the mapping mechanism. If execution is to begin with the K^{th} instruction of P1, the instruction location registers are set as follows:

SOURCE 0	PARAMETER	PAGE	LINE
α	0	0	K

The associative memory represented by the table uses SOURCE, PARAMETER, PAGE as argument and selects the first line. The physical block number 6 is concatenated with the line number (K) to make the physical memory address 6,K from which the instruction is fetched. The parameter space name α from column 4 is stored in SOURCE 1.

Subsequent operation depends on the nature of the instruction fetched from memory at address 6,K. Three cases will be examined to illustrate the operation of the address translation map.

Case 1. A typical instruction for which the operand is the content of memory at the address specified in the instruction. In this case the address translation is identical to that of the instruction fetch described in the paragraph above. SOURCE 0 remains α, and the address from the instruction is of the form PARAMETER, PAGE, LINE.

Case 2. The same as Case 1 but indirect addressing is specified. Operation begins as before but when the content of memory by the first memory access is used as PARAMETER, PAGE, LINE of the address for the second memory access. However, SOURCE 1 is used in place of SOURCE 0 to select the parameter space for this access. For example, if the address from the instruction is

PARAMETER	PAGE	LINE	
1	0	m	INDIRECT

and the first physical address accessed is 5, m, and the value of SOURCE 1 is β. On the second access the parameter space accessed is β as indicated by the SOURCE 1 register. Indirect addressing can be nested by this mechanism and the parameter space for the next access after the operand is fetched is determined by the content of the register SOURCE 0 which will remain unchanged as α.

Case 3. A transfer of control instruction where the next instruction is to be from a new parameter space. In this case the value in SOURCE 1 is transferred to SOURCE 0 before the next instruction is accessed. For example, if the instruction is fetched from the address the value

SOURCE	PARAMETER	PAGE	LINE
α	0	0	K

of SOURCE 1 will be set to β as before. If the address contained in the instruction is

PARAMETER	PAGE	LINE
1	0	n

the address of the next instruction will be

SOURCE 0	PARAMETER	PAGE	LINE
β	1	0	n

because SOURCE 0 is set to β by the content of SOURCE 1.

Simple extension from these three cases will cover all the addressing cases which are analagous to those of a conventional computer employing indirect addressing and indexing. The common technique of using indexing of greater range than the range of the DISPLACEMENT component of the address field in storage presents no problem.

It is clear that the capacity of an associative memory which it is reasonable to construct is too small to accommodate all the information for all users of a computing system. The same technique employed in other computers (see reference 5) in which the entire map is stored in general memory and only those currently active in the associative memory is employed. Reference 6 gives a detailed description of the structure and manipulation of such tables.

Use of the system by multiple users presents no problem. One can think of users as owning parameter spaces and being identified by the assignment of parameter space names. Some parameter spaces may be common porperty and may be identified as belonging to the system as a kind of pseudo user.

An important characteristic of this mapping mechanism is that each row of the table represents an access path. The use of this characteristic for access control will be described in the next section.

It should also be noted that the hardware of the mapping mechanism described is almost identical to that of other mapping mechanisms employing small associative memories. Its cost and speed is therefore similar to that of the more familiar systems such as that described in reference 5.

Access control

Protection or access control in computing systems is usually of two kinds. Two modes of operation, sometimes called user and monitor, exist and are distinguished by the monitor mode permitting program access to input/output instructions and to instructions which alter the state of the machine; these are called privileged instructions. Access to memory is controlled by restricting access to regions of physical memory. In systems employing memory mapping, access control is applied to units of information called pages or segments.

As was shown in section I.B, the desired memory protection depends on the relationship between pairs of parameters; it is access path protection. A conventional computing system can provide such protection by dynamic modification of protection every time a subprogram is called and by calling the supervisor for each return to perform the inverse action.

In the system described, much better protection may be achieved because the CONTROL entry in each row of the map permits specific control of access on each access path as desired. We will now show different possible ways of protection.

A. If a call does not permit access to a given segment, the segment does not appear as a parameter of the parameter space. Of course, a parameter space may provide access indirectly to other segments through one of its parameters if that parameter is defined as the root parameter of a parameter space with the indirectly accessible segments as its actual parameters. Each level of access can be controlled through such parameter spaces.

B. In each parameter space the access to each parameter may be specified independently; a particular segment may, for example, be an execute only subroutine in one parameter space and a data segment in another parameter space.

C. A program parameter has protected entries if it is possible to give control to only a certain number of privileged points in it. To implement this feature conveniently, one can reserve the first N locations of a parameter to point to N entries in the parameter. A protected branch instruction to this segment must then be an indirect branch through one of these N locations.

In the preceding we have considered the control of access to information. This is an extension of the idea of memory protection. Let us now examine use of the same mechanism to control access to processing resources. This is an extension of the ideas of privileged instructions. In the familiar systems all instructions which under any circumstances may be hazardous for a user to execute directly are permanently forbidden to all users. Thus, for example, no user may ever make direct use through his own program of any input/output equipment or secondary storage equipment. The *access path control* concept permits the same kind of flexibility of control to peripheral equipment and instructions as to information. For example, in a particular parameter space on a particular access path a user may be given access by means of his own procedure to a particular magnetic tape unit with no hazard to users of other tape units. More generally stated, the access to a particular instruction, peripheral unit, or other system capability depends on the relationship between the access path and the capability.

It may be impractical to reserve sufficient space in the CONTROL field of the map to permit the complete range of flexibility described above. A reasonable implementation is to have reserved in general memory some space corresponding to each row in the table and a single bit in each CONTROL field, which when set to 1 causes the hardware to fetch this additional control information. Control information relating to such resources as peripheral equipment for which maximum speed is not essential should be stored in this reserved space in memory.

In all these cases we have considered parameters instead of segments. A segment can be a data parameter for one parameter space and a program parameter for another. A program segment can have protected entries for one parameter space and no protection for another. The real power and perhaps the justification of the system lies in the fact that protection is provided in the parameter spaces and not by segments, as has been done in other hardware systems. If we return to our example of a debugging program DDT and its object program P, one can conceive the the following structure:

α: (DDT P.RO., β, D1 D., D2 D.)
β: (P P.RO, D1 D., P2 P.RO.PE)

with the conventions,

P. for program
D. for data
RO. for read only
PE. for protected entries.

In this hypothetical situation D1 is a data segment on which P is working; D2 is a segment of tables about P which is used by DDT; P2 is a subroutine used by P and which will be supposed to be already debugged.

Consider as another example an executive program which handles input/output of files through buffers.

Most of these operations are not really input/output, but exchanges between buffers and the user programs. Such an executive program requires only limited use actual input/output instructions. It should call a subprogram having capability to use appropriate input/output instructions when real input/output is involved. The protections needed here are:

A. The user program is not allowed to jump everywhere in the executive program.
B. The number a user gives must be checked as being a real file name belonging to that user.
C. Access path protection must be provided to input/output and peripheral equipment.

The first protection is given by protected entry. The entries correspond to the different kinds of input/output, for example character, work, block, number, string, line, and so on. The second protection can also be provided easily. Suppose the file number consists of the segment number (in the user space or in the parameter space of the executive program) of the segment which constitutes the buffer for this file. The executive program is a given parameter space in the user program. Each buffer appears for the executive program as a parameter in its parameter space. If the file number the user gives is not really a file number of the user, it will not appear in the parameter space of the executive and an instruction trap will result. This kind of protection frees the executive from such checks. More sophisticated protection can be given in the case where the system provides many input/output packages. If a certain file has been opened by a certain package in such a way that it will not make sense to other packages, the corresponding parameter will not appear in the parameter spaces of other packages. This does not complicate the logic of the program which handles the memory traps, because the same kind of situation can occur anywhere in a user program and must be serviced. Because errors on file numbers occur quite infrequently, an automatic protection saves much time and programming.

Subroutine calls and transmissions of arguments

The call of a subroutine usually corresponds to a change of parameter space. The parameter space is not always completely defined before the call. To complete the definition some information may have to be transmitted from the calling routine to the called routine. This process transforms formal names in the called space to actual names or to values. At the time of the call it is often necessary to store some information about the calling segment in order to be able to return to it. This information is called the "link."

A. Links

A link consists of the address of the call. The complete address in user space which identifies the parameter space and specifies the logical address in the parameter space must be stored. Users must not be permitted to alter directly the parameter space name in a link because an incorrect parameter space name (source) may permit unauthorized access to the executive or to the computing processes of other users. Links are stored in a protected stack type segment which is attached to each process. A link is pushed into the stack and popped from the stack by a return.

B. Transmission of arguments

An argument of a subroutine may be an entire segment, a portion of a segment, or a value. The mapping we have thus far described is adequate for the first two categories. It can, however, be extended for the other category. When transmitting arguments it is necessary to transmit information about their addresses as well as the capabilities attached to these arguments. These capabilities may be specified by the map of the calling routine and by the formal parameter in the new parameter space for the called routine. For the case where there is conflict between the two specifications, some rule has to be designed to determine the resultant capability of the actual parameters.

Consider now the three types of arguments:

1. *Entire segment*

The argument is equivalent to a parameter as defined previously. The call is

$$P(i1, i2, i3, \ldots in)$$

where ij is the number in the calling parameter space of the parameter number j in the called parameter space i (ij = 0 means no parameter transmission). At the time of the execution of the call, the processor scans the parameter list and copies the information relative to the parameter ij from the mapping table of the old parameter space to the mapping table entry for parameter j in the new parameter space; this information is composed of parameter space number plus the capability.

2. Portion of a segment

It is often desirable to transmit only a portion of a segment as a parameter. If this can be done, greater capability may sometimes be given to the access path to that portion than could be given to the entire segment. Such an argument should look like any other parameter. Consider a segment A composed of a vector of vectors. Suppose we call on a routine B which operates on one vector only. The user may give a new segment name to the selected vector which will then be handled by the standard mechanism.

3. Values

Sets of values can be stored in a stack segment and accessed directly.

CONCLUSIONS

The system described provides *access path control* for access to information and to processing resources in a multi-access interactive computing system. Three benefits of this are (1) strong selective control of access to information, (2) dynamic binding capability at run time, and (3) elimination of arbitrary restrictions on access to peripheral equipment and processing instructions. These benefits are all implemented by one simple mechanism. The system also provides for execution of a hierarchy of computing processes having controlled interaction but executed independently in time.

Although the system has not been implemented, the hardware and the software problems have been analyzed extensively as reported in reference 6. The improvements are gained without substantive increases in hardware cost, and the software complexity is reduced.

Many of the system concepts have been developed by others. What is claimed is an integrated system design with a reasonable set of hardware-software decisions, which is an improvement over previously described systems.

REFERENCES

1 T KILBURN
One level storage system Trans of the Professional Group on Electronic Computers 2 223 1962

2 R S BARTON
A new approach to the functional design of a digital computer Proc Western Joint Computer Conference 1961

3 J P ANDERSON
A computer for direct execution of algorithmic languages Proc Eastern Joint Computer Conference 20 184 193

4 J B DENNIS
Segmentation and the design of multi programmed computer system IEEE International Convention Record Part 3 214-225 1965

5 E GLASER J COULEUR G OLIVER
Description of the GE 645 Proc Fall Joint Computer Conference AFIPS 1965

6 J Y LECLERC
Memory structures for interactive computers
PhD dissertation University of California Berkeley June 1966 or project GENIE document No 40 10 110 University of California Berkeley May 25 1966

7 B W LAMPSON
Reference manual time sharing system project GENIE document No 30 10 31 University of California Berkeley August 1 1966

The Air Force computer program acquisition concept

by MILTON V. RATYNSKI
Electronic Systems Division L. G. Hanscom Field
Bedford, Massachusetts

INTRODUCTION

The classic approach to the development of Air Force operational computer programs has been to award separate contracts for the hardware and for the "software" aspects of an electronic system. The integration of the hardware and the "software" into a homogeneous total system has not always been a planned certainty, primarily due to the lack of definitive management procedures and contractor guidance and control techniques for the area popularly known as "soft-ware." It was with this history in mind that a study[1] was initiated to evolve a technique which would permit the following:

a. Development of contractual requirements standards to provide positive control over the contractor's effort in the development of computer programs.
b. Development of a series of principles and procedures which would be applied to the configuration management of computer programs.
c. Development of a series of design standards and specifications for use during computer program design and development.
d. And most important of all, to develop an orderly process of "software" production that would assure full compatibility and timeliness with hardware production.

The "development" and "acquisition" of computer programs, as discussed in this paper, has been broadly categorized to two areas:

a. Those early events, activities and steps involved in INITIALLY acquiring the computer hardware and the related computer programs. Here is the arena for activities and problems from the viewpoint of the system manager, the system designer, and the system acquisition agency.
b. Those events and activities by the military user and implementor to maintain and constantly update the computer programs and the computer program data AFTER the initial acquisition. In this instance, the activity is primarily with the "software" implementation process from the point of view of the "software" implementor's problems within his own environment.

This paper and the series of papers to follow address themselves to area "a," that series of events and activities which involve the initial design and acquisition of computer programs and the integration of the resultant efforts into the overall system. Those activities identified in the second category, "b," are the user's activities after the system is declared operational and turned over to the user. The "b" activities involve the updating and maintenance of existing operational computer programs and the evolvement of a limited number of new and additional programs. Management procedures for the "b" activities are in the process of evolvement by the Electronic Systems Division for the U. S. Air Force and will be presented at a later date.

The objective of this paper is threefold:

1. To introduce the Air Force developed procedures for the acquisition and control of command system computer programs based on the concept of the computer program as the end item of a definable process;
2. To suggest that the developed process is general enough to cover a wide variety of users outside the military domain; and
3. In terms of this process and the Air Force system management philosophy, set the stage for the papers[2,3,4] that follow.

The computer program as an "end item"

Classical definition of "software"

The term "software" is being removed from official Air Force terminology because of the diversity of its entrenched definitions; some of the more popular definitions of "software" are as follows:

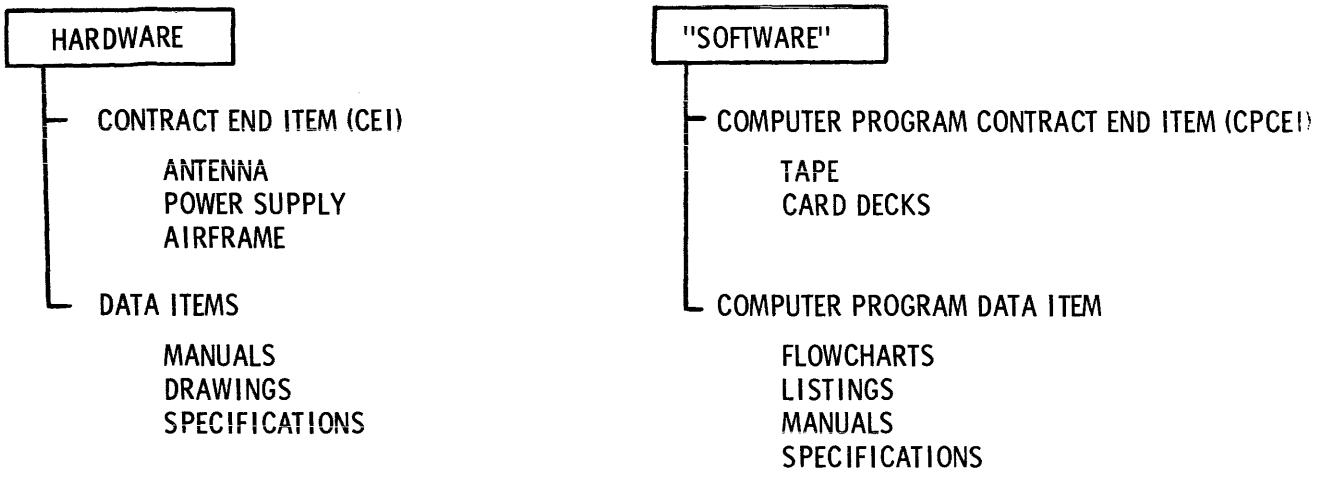


Figure 1 – The computer program as an end item

1. Computer programs, together with associated data, training materials, and the operational personnel.
2. All computer programs, including the hardware of the computer system itself.
3. All computer programs, excluding hardware.
4. Special utility computer programs which are supplied by the digital computer manufacturer as part of a computer.
5. Deliverable contractor data associated with the development and operation of system equipment.
6. System personnel, as contrasted with system hardware.

"Software" as an end item

In order to adapt the Air Force 375 concept of system management to "software," it was first necessary to define "software" precisely and, more specifically, to attempt to describe it in engineering and hardware terminology. It was determined that it was more reasonable to adapt "software" to the "375 hardware acquisition techniques," than to develop a separate "software" management process, which in turn would require a third or higher level management process to integrate "software" and hardware into a total system.

Fundamental to the management of "hardware" is the readiness with which it is possible to identify an item of hardware as an End Item (a Contract End Item[5]) and the various paper products (Drawings, Manuals, Specifications) as Data Items. It is this philosophy which was adapted to "software," and as depicted in Figure 1, "The Computer Program as an End Item," permitted the identification of "software" products into the more precise categories of:

(1) Computer Program Contract End Items (CPCEI), and

(2) Computer Program Data.

It will be noted from Figure 1 that, as is the case with hardware, the computer program is now a designated, deliverable, and contractually-controlled product. The Data Items for both hardware and for computer programs are those paper products which "describe" the deliverable "end item" or describe how to use, test, or update the end item. Card Decks, although physically "paper," are essentially integral to the direct operation of the computer itself, and are not considered a Data Item per se. Both the hardware and the computer program data items have standards for their preparation, however, the format and content for computer programs is recognized in that there are separate "standards" for hardware and separate "standards"[6] for computer program data items.

Definition of contract end item (CEI)

The term "contract end item" (CEI) has been selected to identify individual design packages. A contract end item has become a very useful and common reference for program management, i.e., alignment of design responsibilities, contracts, schedules, budgets, etc. The CEIs, thus, represent a computer program or a level of assembly of equipment selected as the basis for management. Once the "design package" has been identified as a contract end item (CEI), this end item is then described in an individual specification.[2]

Experience to date with electronic command systems contracts,[7] has indicated that there is indeed a common basis for management when computer programs are designated as CEIs; management by the government of the contractor's efforts and management by the contractor of his own or subcontractor's efforts. An understanding by both the government and the contractor of what is to be delivered is an important step towards developing an orderly process for the production of computer programs.

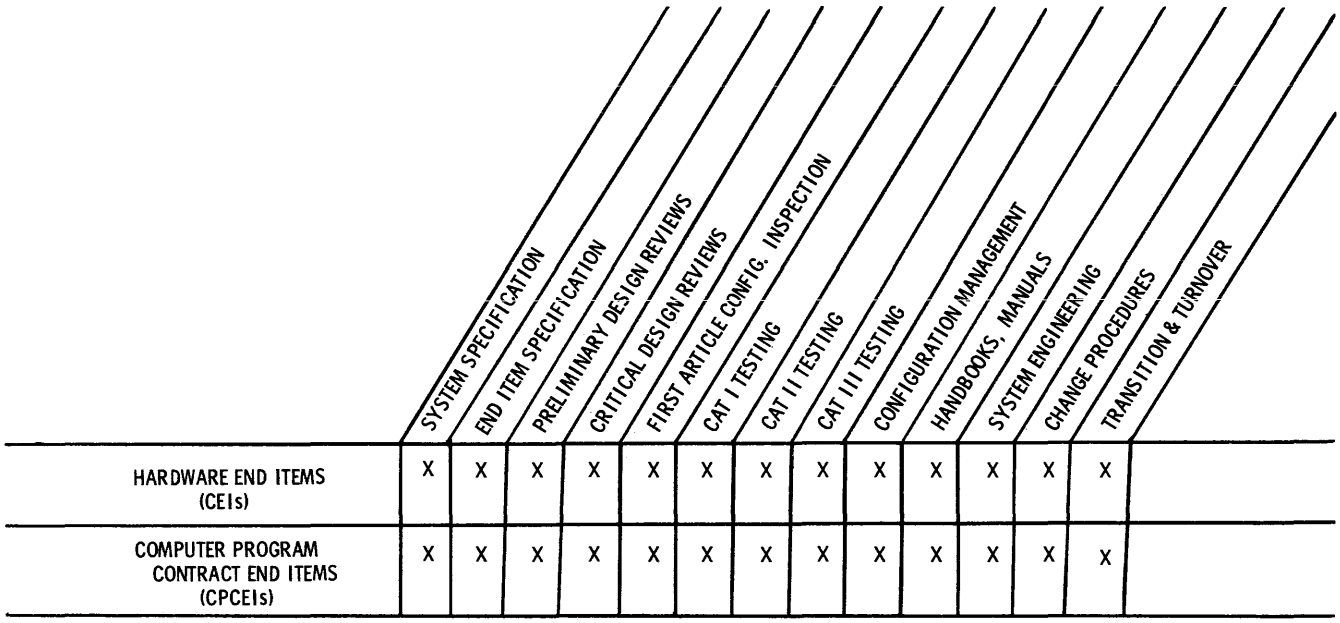


Figure 2—Some commonalities between computer program end items and hardware end items

Computer programs and the 375 system management process

The basic decision that a computer program could be defined as a Computer Program Contract End Item (CPCEI) permitted adaptation of virtually all the Air Force 375 System Management concepts to the management of computer programs. Specifically, the concepts of uniform specifications, baseline management, change control, specification maintenance, design reviews, and a test program became all as pertinent to computer programs as they are to hardware acquisition. More importantly, however, a common definition and acquisition process now assures a full integration and in a timely manner of both the equipment and the computer programs into an integrated and homogeneous military system.

There were sufficient, substantive differences between hardware end items and computer program end items to warrant the development of separate but companion procedures and standards for computer programs, and these will be described at opportune moments during this and the following papers.[2,3,4] However, it was possible, in the real live world situations that the Electronic Systems Division is constantly exposed, to adapt the "end item" philosophy of management to computer programs, and yet retain full technical integrity and full technical identity for all disciplines concerned.

Comparison of computer program end items with hardware end items

Figure 2, "Some Commonalities between Computer Program End Items and Hardware End Items," portrays in a capsule form that the major technical events and milestones, which pertain to hardware, are also applicable directly to computer programs. Under the Contract End Item (CEI) concept, there is a common and understood method for specifying the technical requirements prior to computer program design, for specifying the interfaces between computer programs/hardware/personnel, and for the development of sufficient computer program documentation to assure that the user can operate the system effectively.

The basic premise is that it is equally important to specify the system requirements for computer programs as it is for hardware, thus the initial common ground for both is in the System Specification. In the normal design process for military or for commercial computer programs, both the hardware designer and the computer program designer require a document identified as a Specification, in which the performance/design and test requirements and/or parameters can be detailed. Likewise, design reviews, testing, change procedures, manuals, technical descriptions of the product are equally common to both areas. The discussion on the system management process that follows attempts to point out that it is primarily due to the "commonality of approach" by the computer program designer and the hardware designer, that it was possible to adapt the 375 process to computer program development.

Just as it was possible to identify common activities, it became quickly apparent that substantial differences

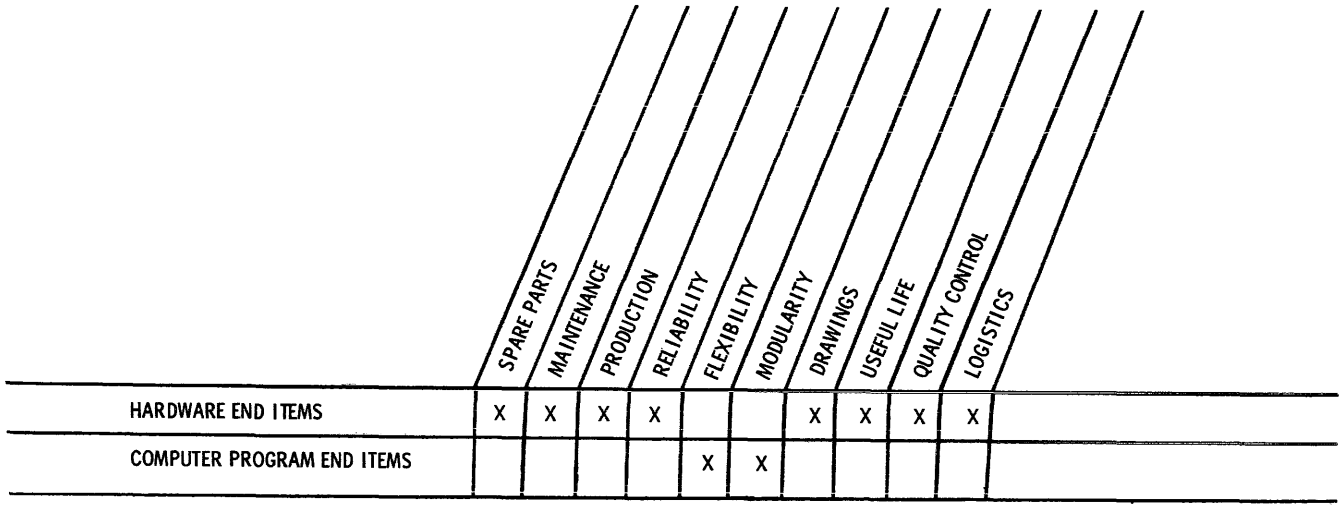


Figure 3 – Differences between hardware and computer program end items

existed too. These are listed in Figure 3, "Differences between Hardware and Computer Program End Items." Unlike hardware, computer program instructions do not "wear out." Further, it is not necessary to build and to maintain an elaborate special production facility to produce each computer program in quantity since, regardless of its configuration, any number of copies can be normally duplicated on standard machines at small cost. Therefore, the elimination of concepts, such as, reliability, spare parts, quality control, useful life, etc., required careful tailoring of the hardware oriented management procedures to computer programs. It is felt that this goal was substantially achieved, and the following discussion on the pertinence and relative compatibility of the Air Force 375 System Management Process to computer program development is an attempt to demonstrate that, with modified procedures, the basic 375 process can readily be accepted by the computer program community.

Concept of systems management

The contemporary Air Force 375 series regulations and the AF Systems Command manuals establish the procedures of acquiring a new military capability in a finite and structured manner. These documents were intended to include all the activities and all the events necessary to assure a complete military system, however, the emphasis in the current AFSC manuals is on hardware acquisition and does not adequately identify the inherent peculiarities of "software" acquisition. The "375" technique, nevertheless, can be applied and the "software" activities readily associated with the 375 life cycle. This is the objective of the discussion following.

The system life cycle is divided into four phases: Conceptual, Definition, Acquisition, and Operational, as shown in Figure 4.

An overview of the Air Force system management process

System management of large-scale military systems is a complex team effort, in which many government agencies and the contractors pool their resources to balance technology, cost, time and other factors to achieve the best performing product for the military inventory. One of the most important points to understand about the 375 process is that it is *not* a new way of life nor is it an entirely new approach to the development of a system. As stated by Gen. B. A. Schriever,[6] the Air Force is taking advantage of the lessons learned over past years, and has formalized and standardized on a "typical" way of doing business. This is not to say that every program or project was or is intended to follow all the 375 procedures or all the processes, but that it is a "road map" of events, activities, and milestones that make sense, can save the taxpayers dollars, and the military time. The emphasis in the Air Force has always been on the "selective use" and on the "discrete application" of these formal and standard procedures."

The final portion of this paper is essentially an introduction of a management concept for computer programming and is intended to set the stage for the papers[2,3,4] that follow. The significant activities of the programming process, in terms of the four phases of a systems' life cycle with an emphasis on key milestones and key outputs, are summarized on a phase by phase basis.

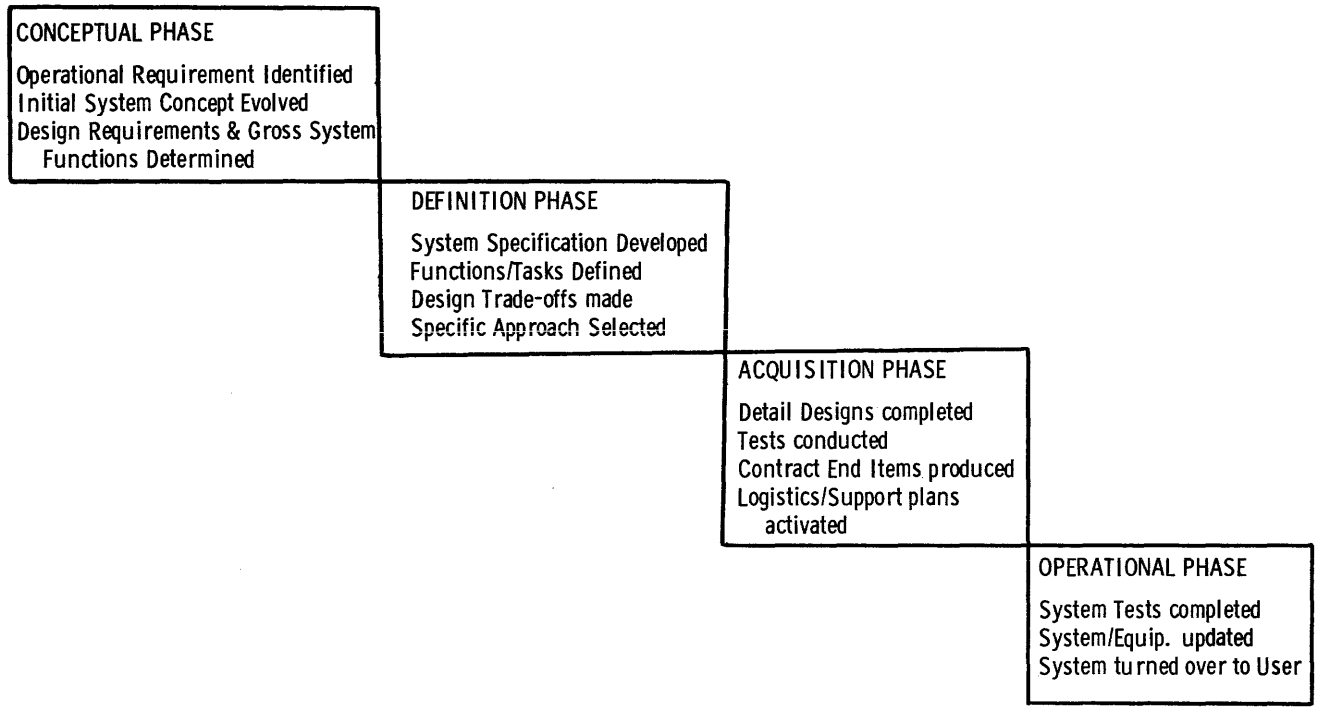


Figure 4—Air Force system life cycle

The conceptual phase for a computer program

This is the period during which new inventions, technological advances, user requirements, time and money, and other considerations are traded off to define a valid requirement.

Of particular interest to computer program development is a firm delineation of the system mission, the operational environment, interfaces, and doctrine of operational employment. For information management and handling systems, the analyses of the system information processing functions should begin during the Conceptual Phase. These analyses will typically include the detailing of system functional flows, the identification of design requirements, and the performance of trade-off studies based upon estimates of cost effectiveness.

Figure 5 depicts typical key events/activities that would be important to insure early and adequate

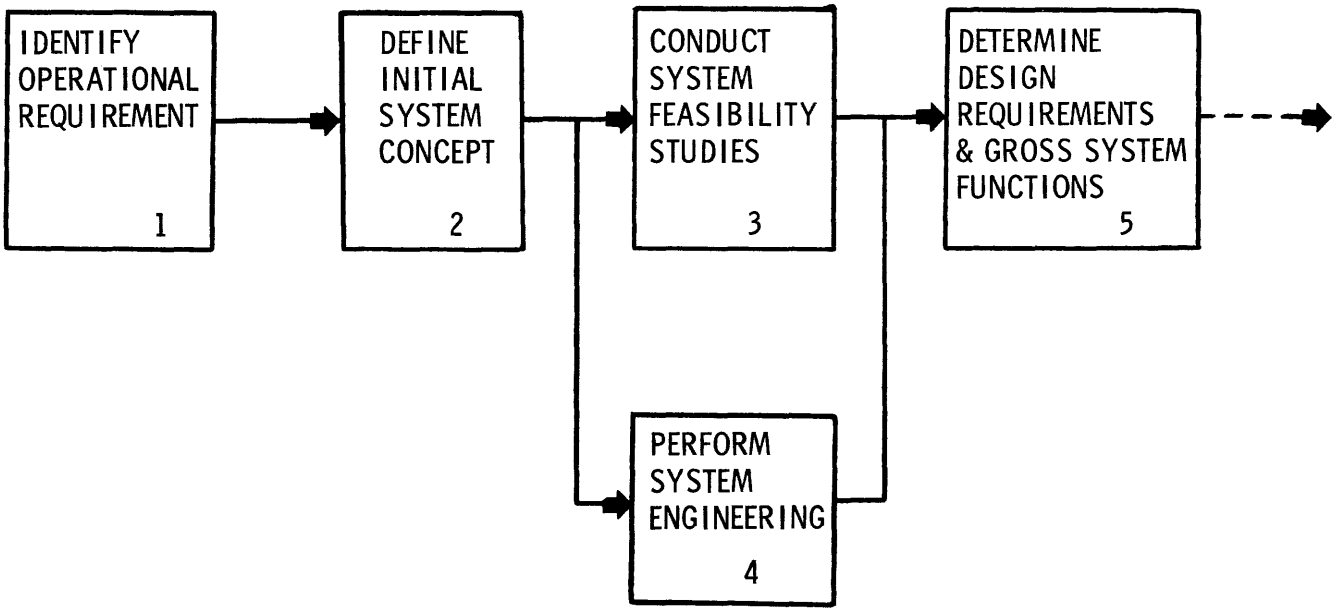


Figure 5—The conceptual phase for a computer program

consideration for the computer programming aspects of an information processing system.

Block 1, "identify operational requirement"

For computer-based systems/subsystems, this is the initiation of the information processing analysis. Mission requirements are analyzed, the operating concept is defined as are the modes, environments and constraints. The responsibilities, geographic locations and the levels of the user are identified. A preliminary estimate of personnel requirements is made. Depending on whether the requirement is to satisfy a military or a non-military user, the information processing requirements are included in the appropriate planning documentation, in order to identify the total system operational requirements.

Block 2, "define initial system concept"

This step is concerned with developing qualitative requirements of sufficient potential value as to warrant further effort and definition in specific terms. The qualitative requirements should recognize the status of the pertinent state-of-the-art and should conceive those concepts which will satisfy the basic mission and plans. The initial operational functions and concept are developed at this time, as would be the identification and nature of the sources and the destinations of data elements. The requirements for data collection, generation and/or transmission are initially defined. It is often pertinent to evolve several information processing concepts, in order that various approaches can be assessed, particularly from the major time and money trade-off viewpoints.

Block 3, "conduct system feasibility studies"

This step represents the initial effort toward the formulation of a specific operational requirement for a system. For computer-based systems, this involves the translation of a qualitative requirement into system concepts by the assignment of functional and operational capabilities to specific areas of equipments, procedures, and man/machine relationships on a trial basis. This is the first point at which the considerations of timeliness, technological and economic feasibility, and capability are explored in any detail.

Block 4, "perform system engineering"

A specific concurrent and prime technical activity at this point in time is the system engineering activity. In the case of military systems, this activity will be performed more and more by in-house government agencies and less and less by contractor-industry activities. For the Air Force, captive contractors, such as, the MITRE Corporation and the Aerospace Corporation, and in-house engineering agencies, such as the System Engineering Group and the Rome Air Development Center, perform the bulk of the initial system engineering studies.

System engineering studies are conducted to the level of detail necessary to establish that the system selected will meet the operational performance capabilities and will be technically attainable within the required time period. Basic technical data should essentially be complete in the following areas:

a. *Information processing.* This will at least include: a description of the mission requirements, including deployment requirements and integration with other systems; concept of operation associated with each mission/environment; the system information flows; the allocation of functions between man and computer; a description of the system data base including processing and storage requirements; a description of selected techniques of computer programming; a description of the command organization with preliminary estimates of personnel requirements; a description of requirements for system exercising; and a description of interfaces with other information systems.

b. *Computer and associated equipment.* The performance characteristics of the computer, the operational consoles, the other Input/Output units, and the special equipments for system exercising are derived from the analysis of system information processing functions, in parallel with the determination of functions to be performed by personnel and computer programs. Both off-the-shelf and state-of-the-art equipments are considered. Additionally, custom-configured or significantly modified existing equipment are included in the cost-effectiveness studies, in order that technical feasibility, cost and schedule estimates, and interfacing characteristics with the computer program system, communications, and facilities are firm enough to warrant the development of a System Performance Specification. Factors, such as, general, logical and physical equipment configuration; data processing speeds, storage requirements, input-output interfaces; peripheral requirements for magnetic tape units and card machines; operator consoles; special equipments; and growth potential, are identified preliminarily at this point in time in order to assure feasibility of the system to meet the mission requirements.

Block 5, "determine design requirements & gross system functions"

This step signifies that sufficient system concept studies, system feasibility studies, exploratory development, and/or system engineering have been

performed to satisfy decision makers that the system costs are realistic, estimated schedules realizable, the high risk areas identified, and that this is the system which should be funded and resources allocated with which to proceed to the next phases, the Definition/Acquisition Phases.

The definition phase for a computer program

This phase, which came into being under Secretary McNamara, sees the entire system defined and identified down to the level of contract end item specifications. During this phase, the systems analysis activities are carried out, considering such factors as total system cost/schedule/performance, identification of interfaces and high-risk areas. The Definition Phase culminates with the preparation of a definitized development contract setting forth the results of the Definition Phase as the Design Requirements Baseline.

For electronic systems, the information processing aspects are identified as a system segment. Technical studies and alternative approaches are made at this time in the areas of computer programming, equipment, communications, facilities, personnel, procedural data, and training, including considerations of development lead times, system testing criteria. The direct objectives of these technical studies are to establish the total spectrum of design requirements and design constraints firmly, both for hardware and software, at the levels required for the System Specification and other documents which must be prepared.

Figure 6 identifies some of the major events, activities, and output products that would normally be expected to occur whenever a system goes through the Definition Phase.

Block 6, "initial system specification"

This step represents the technical effort that is applied to preparing a specific document upon which many subsequent steps and processes are dependent. The initial system specification is largely oriented to the operational requirements and allows considerable latitude in the system design, except in those areas wherein gross design considerations are specified or obvious. In some instances, expediency does not permit more than a cursory refinement of previous concepts and tentative performance requirements in the formulation of this document.

The preparation of the initial System Specification generally first requires that trade-off studies are identified and performed and that technical studies, consisting of reviews, verification, and expansion or alteration of technical concepts and data resulting from the activities from the Conceptual Phase are accomplished. If time precludes in-house accomplishment of these studies, this activity may be tasked upon the Definition Phase contractor(s). In this case, the Definition Phase contractor(s) would be required to evolve a fully defined System Performance Specification as one of his end products. For computer-based systems, information processing functions are re-examined and verified in relation to the user's organization, mission, and operating doctrines. Alternative approaches in the areas of computer programming, equipment, communications, facilities, personnel, procedural data, and training are assessed. The direct objective of these studies is to establish

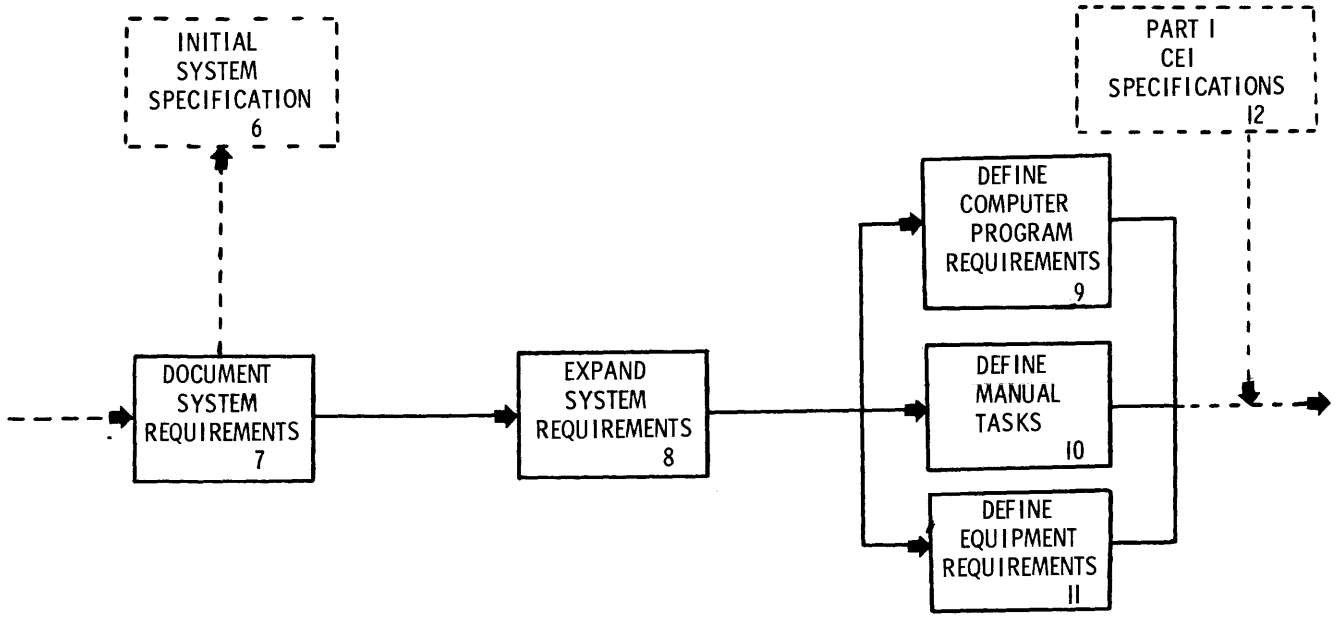


Figure 6—The definition phase for a computer program

the total spectrum of design requirements and design constraints at the levels required for a System Specification.

Block 7, "document system requirements"

Under the 375 system management concept, the entire technical control of the system is based on the establishment of three (3) baselines/specifications. The first of these is the Program Requirements Baseline, the technical requirements of which are defined in the System Specification described earlier. The Program Requirements Baseline consists of documents which provide a conceptual technical description of the system, a description of the Definition Phase inputs/events/expected outputs and the estimated cost. This is the document that is evolved prior to initiation of the Definition Phase. From the total system approach, it is critical that the entire man/machine concept be sufficiently detailed particularly from the time, money and integration aspects.

Block 8, "expand system requirements"

The Definition Phase contractors first technical activity is to conduct a comprehensive and critical reveiw of the functions, the performance, and the design requirements contained in the System Specification. Objectives are to verify, expand, and/or alter on the basis of the contractors approach and technical experience. Selected trade-off studies would be performed, interface requirements of the major hardware and computer program end items would be expanded.

Block 9, "define computer program requirements" and Block 11, "define equipment requirements"

The nature of the technical efforts will typically begin to diverge at this point in time with respect to their direct concern with system operational functions. This effort will define the requirements for the detailed tasks to be performed by the operational computer programs and the system equipment. It is predicted on the basis that the interfacing equipment characteristics have been defined at a level which will permit the analysis and specification of functions to be performed by the computer programs and equipment contract end items.

Block 10, "define manual tasks"

A concurrent activity is the definition of manual tasks. The technical effort during this phase encompasses the analysis of manual and man/machine tasks. This leads to the definition of:

a. manual tasks and procedures
b. console operator procedures and decisions
c. design requirements for console controls and displays.

Block 12, "Part I CEI specifications"

One of the most important products of the Definition Phase is the development of all the necessary "design-to" specifications for the Computer Program Contract End Items. In Air Force terminology, these are called Part I—Performance and Design Requirements Specifications.[2] This part of the CPCEI specification is used to specify requirements peculiar to the design, development, test and qualification of the contract end items (CEIs). The Part II—Product Configuration and Detailed Technical Description specifications[5] are not prepared until well into the Acquisition Phase. An important section of every computer program Part I specification is the detailing of the requirements imposed on the design of a computer program end item because of its relationship to other equipment/computer programs. Interfaces defined in this section of the specification shall include at least all relevant characteristics of the computer, such as memory size, word size, access and operation times, interrupt capabilities, and special hardware capabilities. The Part I specification permits the first opportunity for initiating configuration management procedures[5] for the control of design requirements changes. Part I specifications permit competition for the Acquisition Phase contract(s).

The acquisition phase for a computer program

The contractor designs, produces and tests the system hardware and computer programs during this phase. This phase continues until the last article is delivered to the operational user and all system development test and evaluation has been accomplished.

Figure 7 identifies many of the important activities in the design and validation process that would normally be initiated during an Acquisition Phase.

Block 13, "Part I specification"

Now that the initial end item performance specifications have been developed, the test requirements detailed, and all the necessary man-machine tradeoffs made, it is necessary that the System Specification be updated and approved for contractual use in the Acquisition Phase contract(s). For hardware, it is usually possible to have defined in detail the functional interfaces, however, for computer programs this is normally not possible unless a firm selection and approval of specific hardware, computer programs, and communications has been made. Thus, the system specification will normally not be fully updated, insofar as computer program requirements are concerned at this point in time, but rather, after

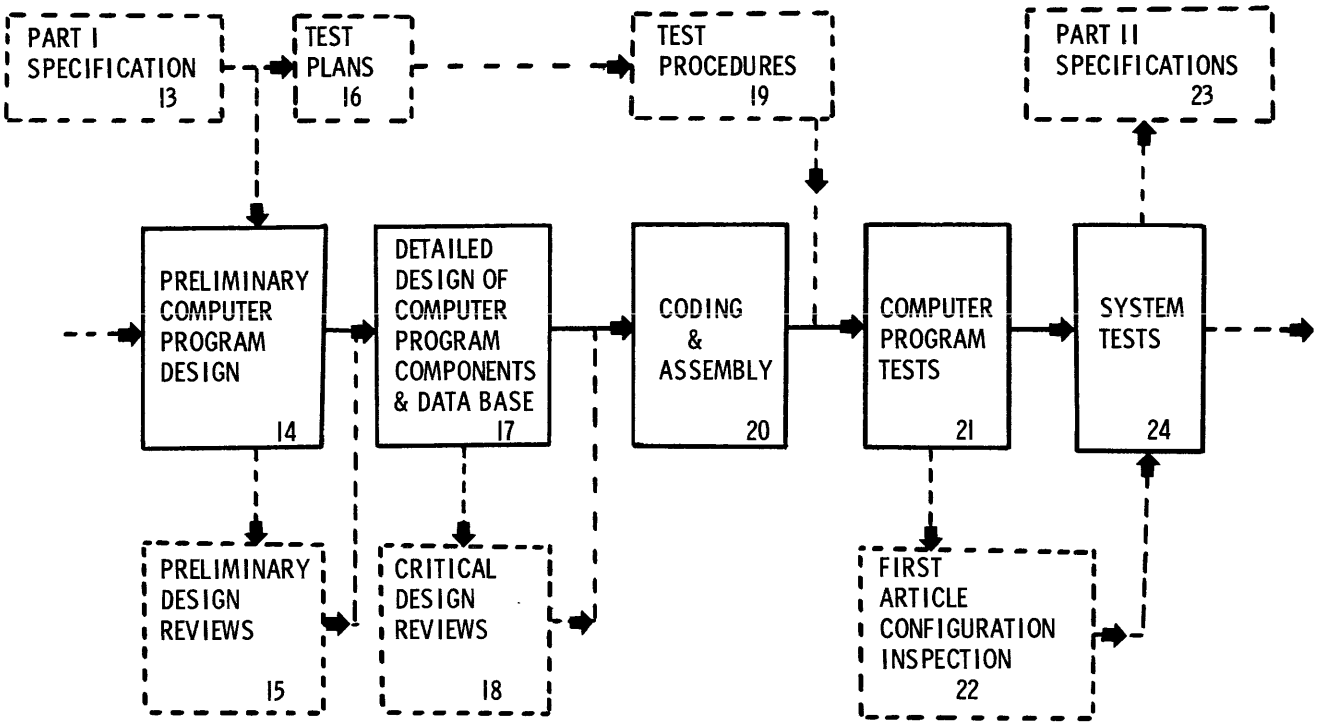


Figure 7—The acquisition phase for a computer program

some of the Design Review activities in the Acquisition Phase.

Blocks 9, 10, 11 and 12 constitute the technical substance of the Design Requirements Baseline. This baseline represents the technical requirements for the Acquisition Phase contractor(s).

Block 14, "preliminary computer program design"

The fundamental purpose of the Acquisition Phase is to acquire and test the system elements required to satisfy the users requirements. Upon award of the Acquisition Phase contract(s), the design of all contract end items begins. The Part I (design-to) specifications developed earlier, plus the System Specification, will be the contractor's technical baseline and, essentially, are a start on the design of the end items. For computer program end items, this activity consists of finalizing the analysis of system inputs/outputs/functions, allocation and grouping of computer programming tasks into individual computer programs or program packages. Additionally, individual programs will be described and the interprogram interface requirements detailed. All this, plus the development of functional flows, the accomplishment of trade-offs, etc., will be the basis of the detailed design that will be described in the Part II specification.

Block 15, "preliminary design reviews," Block 18, "critical design reviews," and Block 22, "first article configuration inspection"

During the Acquisition Phase, a series of technical reviews and inspections are held to provide discrete milestones for an exchange of technical information between the Air Force and the contractor. Two of these, the Preliminary Design Review and Critical Design Review, are analyses of the design of the contract end items at various stages of development. The third milestone, the First Article Configuration Inspection, is an audit of the documentation that describes the contract end item. A more detailed discussion of these milestones is contained in the paper by Piligian.[3]

Block 16, "test plans," and Block 19, "test procedures"

A parallel effort to the computer program end item design is the development of a Category I qualification test program. Draft of the test plan and associated test procedures for the validation and evaluation of computer program end items are written for submission to the procuring/engineering agency for approval.

Block 17, "detailed design of computer program components & data base"

The detailed design of the computer program is accomplished based on the approved Part I Specification and the results of the Preliminary Design Review.

The detailed flowcharts logic diagrams, narrative description, etc., will provide the basis for actual coding of the computer program(s). The design of

the data base is developed at this time resulting in the number, type and structure of tables, as well as description of the items within the table. The detailed design forms the basis for the Critical Design Reviews[3] discussed previously.

Block 20, "coding & assembly," and Block 21, "computer program tests"

Immediately following the Critical Design Review, coding of the individual components of the computer program end items takes place and the process of checkout and testing of the components begins. Computer Programs are generally first tested during the Category I^3 test program. Cat I demonstrates that the CPCEI, as produced, satisfies or does not satisfy the design/performance requirements of the Part I CPCEI specification.

Block 23, "Part II specifications"

The Part II Specification is a detailed technical description of the CPCEI. Essentially, it consists of elements which have appeared as computer program documentation in the past in a variety of forms and combinations e.g., functional allocations, timing and sequencing, data base characteristics and organization, flowcharts, etc. The Part II Specification organizes these into a single, comprehensive description which includes the data base and listings, It is an "as-built" description, which the procuring agency accepts only after its completeness and accuracy have been formally verified, and normally after the computer program performance has been qualified in relation to the Part I Specification. Like the Part I, it does not contain operating instructions, test procedures, or other non-specification materials. Its subsequent technical uses are as an aid in correcting errors and designing changes. Following formal verification of its completeness and accuracy, it defines the Product Configuration Baseline.

Block 24, "system tests"

The objective of Category II system tests is to demonstrate that the system will satisfy the system performance/design requirements of the System Specification. For computer programs, Category II testing will assure the overall systems engineer that the computer programs are fully compatible with the hardware in an integrated performance environment, and that they meet the total system requirements, including personnel, communications, man-machine relationships, etc.

The operational phase for a computer program

The Operational Phase permits the transfer and turnover of systems/equipments from the design and acquisition agency to the using (customer) agency. This phase begins when the operational user accepts the first article and ends when all the elements of the system have been accepted by the user.

Most Air Force systems are turned over to the user on an incremental basis: squadron by squadron for missiles; site by site for electronic systems; or system element by system element. In the case of computer-oriented systems (command/control systems), there is an evolutionary aspect present, which seldom permits the completion of specific Operational Phase. In this instance, it might be stated that there may be many Operational Phases for any one System, as a requirement, for system-updating occurs and as this requirement is satisfied.

As the development agency acquires end items, and satisfies the user that a group of end items meets the user's operational requirement, a formal release of both the physical entities, as well as the engineering responsibility, is made by the development/acquisition agency. For computer programs, the transfer of engineering responsibility would not normally be to the Air Force Logistics Command (as is true for hardware), but rather would more likely be to the user. There are instances, however, where certain utility and/or maintenance-diagnostic computer programs furnished with, or as parts of, the computer equipment are engineering-transferred to the Logistics Command, rather than the user.

As elements or end items are turned over to the user, the configuration management responsibility also becomes the responsibility of that agency. For purposes of configuration management of computer programs, the updated System Specification and the Part I Computer Program End Item Detail Specification function as the design requirements baseline throughout the Operational Phase.

Figure 8 identifies many of the specific activities that pertain to a process that permits the design and acquisition agency to turn over to the user those completed elements of a system, as the operational requirements are met.

Block 25, "turnover to user"

Under current Air Force concepts, equipment, facilities and computer programs can be incrementally "turned-over" to the user on an operating unit by operating unit basis. The standards for computer program turnovers is in the process of evolvement, however, it is general practice to "turn-over" both the hardware as well as the applicable computer programs on a concurrent basis. Under any circumstance, turnover implies that the minimum operational capability has been satisfied for the total system or

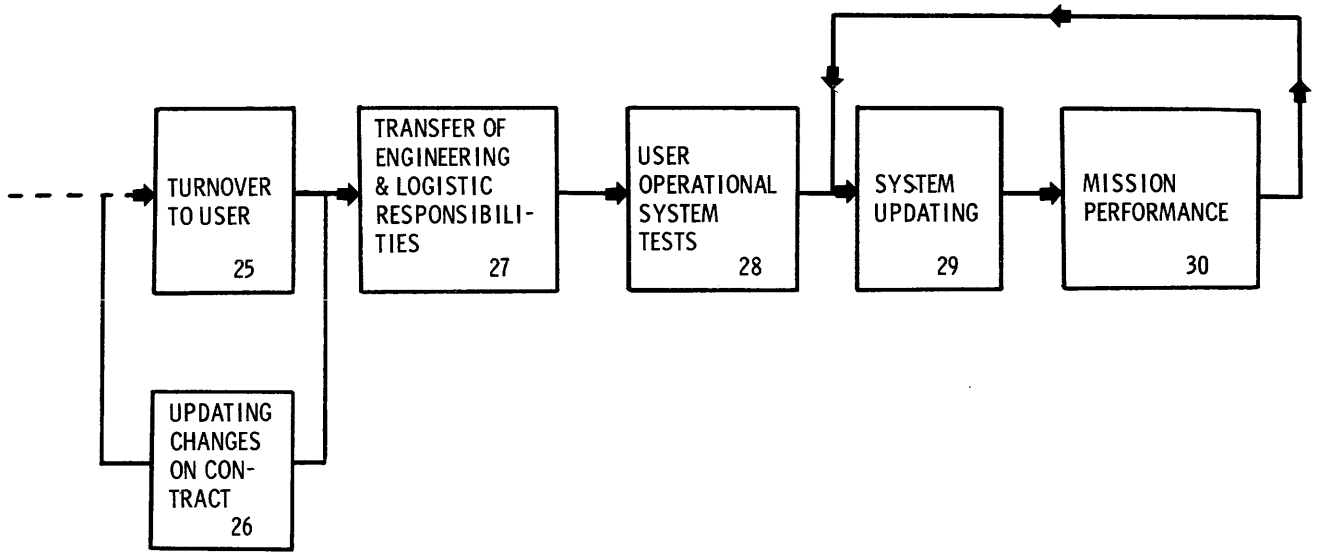


Figure 8—The operational phase for a computer program

for some element thereof, and that the industry and government design agencies consider the primary design/engineering activities as completed.

Block 26, "updating changes on contract"

As with hardware, computer programs are subject to follow-on development tests which normally result in updating change requirements. As the user exercises his system, or some element of a system, deficiencies and/or improvements become suggested. Updating changes are most practical to contracts which have not been completed, and can be broadly categorized as those which are (1) the contractor's responsibility and essentially under contract "guarantee," and (2) improvements and performance updating initiated by the user, which requires a new contractual understanding and new design effort. In essence, the "design-updating" changes that are necessary are now placed on contract, system tested and recorded.

Block 27, "transfer of engineering & logistic responsibilities"

There is no formalized Air Force policy on the logistics and engineering responsibilities for computer programs once they have been turned over to the user, however, an unofficial process is in effect. Several large electronic systems have been officially turned over to using commands. The "engineering" responsibility for controlling corrective actions and incremental improvements of operational capabilities through changes in the computer programs has in all cases, to date, been assigned to and assumed by the using command. In some instances, this responsibility extends to the maintenance of handbooks, user manuals, and other documents which are directly dependent upon the computer program end item configuration. The "logistics" and the so-called "engineering" aspects of computer programs are being studied and debated at appropriate levels, and a decision issued, particularly with reference to certain utility and/or maintenance-diagnostic programs furnished with, or as parts of, the computer equipment.

Block 28, "user operational system tests"

Upon the completion of Category II system tests and the incremental transfer of computer programs, computer equipments, handbooks, user manuals, etc., to the user, the using command may elect to conduct Category III type tests. Category III tests are system tests similar to Category II and for computer programs may either be unnecessary or may be conducted concurrently with Category II. At this point in time, the system is fully operational, all live environment, and is conducted by the user, generally with the assistance of contract services from the automated data industry. As during Category II testing, deficiencies and/or improvements are revealed, and system modifications made dependent on significance to mission and budgets.

Block 29, "system updating"

One of the final technical functions is to document and record the precise system/equipment configuration and to update all manuals, handbooks, etc. Beginning back in the latter part of the Definition Phase, a "configuration management" procedure would have been defined, agreed upon by the contractor(s), and would at this point in time be thoroughly updated and turned over to the user. The user would continue to use the system specification and the Part I de-

tailed "design to" specification, as the design requirements baseline for the purposes of configuration management and for a design updating standard. The system itself is also updated to incorporate updating changes and modifications resulting from the users operational testing and system usage.

Block 30, "mission performance"

This event represents the continued operation of the system by the user until such time as the system is phased out of the operational inventory.

SUMMARY & CONCLUSION

The management framework established by the 375 series Air Force regulations and manuals is applicable to electronic systems, as well as to aircraft and missiles. This management concept is exemplified by electronic computer-based systems, such as, 412L, 416L, 425L and 465L; namely, operational military systems which provide information processing assistance to command personnel who are responsible for planning, decision-making, and control of resources and forces. A major impact of considering this type of system is to emphasize information processing and associated technologies as the lead items in system concept and design. This emphasis is needed in order to assure that the specification and procurement of hardware and facilities will be based upon adequate consideration of the user's information processing procedures and requirements.

The modification of the current hardware-oriented 375 system management process to include "software" activities places primary emphasis on events associated with:

1. Analysis of system information processing functions.
2. Allocation of functions to operational personnel and the computer hardware.
3. Definition of the functions to be performed by computer programs.
4. Development of operating procedures.
5. Design, development, and testing of computer programs.
6. Development of provisions for operational training and other personnel subsystem products.

The implications of the concept described in this paper can be summarized as follows:

1. The normally hardware-oriented System Program Director is now provided with a management tool with which to evolve requirements and assess the total system's progress by the establishment of computer program key events, milestones and activities during the system's life cycle. The "visibility" aspects of computer program intricacies will be more evident to the System managers.
2. Computer program requirements documented in a technical specification provide a standard by which the performance of the contractor and the performance of the resultant computer program can be evaluated.
3. The design documentation and specifications developed during a normal Definition Phase activity will provide for the first time an opportunity for the government to acquire computer programs on a true price competition and open-bid, fixed-price basis, rather than the current predominant sole-source, cost-plus situation.
4. The computer program management techniques evolved will significantly reduce the excessive cost-overruns and will reduce the overall system costs that were uncontrollable without a finite management control process.
5. The eventual user and operator of the electronic system will be provided with sufficient data and documentation which will permit the user to operate, maintain and update the system much more effectively and efficiently.
6. The concept described is considered to be applicable to any system, military, business, or industrial and to any programming exercise that warrants the use of computer program management techniques.

REFERENCES

1 DD-1498:
Computer programming management
Air Force Project 67-2801 1967

2 B H LIEBOWITZ
The technical specification—key to management control of computer programming
SJCC Proceedings 1967

3 M S PILIGIAN J L POKORNEY
Air Force concepts for the technical control and design verification of computer programs
SJCC Proceedings 1967

4 L V SEARLE G NEIL
Configuration Management of Computer Programs by the Air Force: principles and documentation
SJCC Proceedings 1967

5 AFSCM 375-1:
Configuration management during definition and acquisition phases
Air Force Systems Command U S Air Force 1 June 1964

6 AFSCM 310-1:
Management of contractor data and reports
Air Force Systems Command U S Air Force Revisions September 1966 and March 1967

7 System 416-M:
BUIC-back-up interceptor control
Electronic Systems Division U S Air Force

8 B A SCHRIEVER General USAF:
Foreword to AFSCM 375-4—system program management manual
Air Force Systems Command U S Air Force 16 March 1964

Configuration management of computer programs by the air force: principles and documentation*

by LLOYD V. SEARLE and GEORGE NEIL
System Development Corporation
Santa Monica, California

INTRODUCTION

This paper is addressed to Air Force concepts relating to configuration management of computer programs. For this meeting, our purpose is to summarize the nature of the concepts, and to present an introductory outline of the documentation and procedures. Although we are using the Air Force terminology and phased system program relationships, it should be recognized that the same general principles, and many of the particulars, are also in process of being adapted for use by the other military services and NASA. The basic standards lend themselves to general application in the formal management of computer programming projects.

As described earlier in this session,[1] configuration management represents only one of several management areas which apply to a total system program. Contrary to a prevalent misconception, it is not synonymous with, nor a substitute for, the technical system engineering effort which constitutes the heart of a system design and development program. However, as an auxiliary structure which evolves in steps during the Definition and Acquisition Phases, it is closely related with the other areas of systems management, particularly with the processes of system engineering and testing. Its special concern is with procedures for controlling the performance requirements and actual configurations of the various physical parts of a system. One of the principal needs it fulfills is to provide a systematic means by which the system status and technical objectives can be formalized, for the mutual benefit of the many agencies which are typically associated with the procurement, development, logistic support, and operation of a complex system.

Contract End Items (CEIs) are the system parts to which configuration management applies. In the Defense Department and NASA, the CEI designation* does not apply to system personnel, items of data, or to contractual studies and services. It refers only to identified items of equipment, facilities, and computer programs, as the major deliverable elements of a system which is developed for use by an operational agency.

Although the emphasis in this discussion is on computer programs, it should be recognized that the procedures to be discussed are designed to fit within a common framework of configuration management for all types of CEIs in a system. In fact, they incorporate many concepts and terms which have been borrowed from established practice with items of equipment. At the same time, they are specifically tailored to reflect the many unique characteristics of computer programs as contract and items. In general, they are derived from actual experience in developing computer programs for electronic systems, and are presently in use by a number of system contractors.

General concepts

Within the scope of configuration management requirements, distinctions are made among the three major sub-processes of identification, control, and accounting:

Configuration identification refers to the technical definition of the system and its parts, primarily in

*The authors have been associated with the development of computer programming management procedures for Air Force Systems Command, through contract studies conducted in support of the Electronic Systems Division. This report summarizes selected aspects of the Air Force procedures as viewed by the authors; it is not to be construed as an official expression of policy by Air Force agencies or the System Development Corporation.

*In the Army, CEI denotes Configuration End Item, rather than Contract End Item.

the form of specifications. In general, configuration management is based upon the concept of *uniform specifications,* which implies simply that in each system program there should be one general specification for the system as a whole and one specification for each contract end item. General format and content requirements of the specifications are uniform for all systems. However, requirements for the system specification and CEI specifications differ. The system specification is written at the level of performance and design requirements only, while the specification for each major CEI is written in two parts—Part I as a performance-level specification to establish technical objectives and design constraints, and Part II as a technical description of the actual configuration resulting from the design/development/test process. Detailed requirements for specification format and contents are different for the major classes of CEIs, i.e., for equipment, facilities, and computer programs, as a function of the typical differences in technical characteristics. Once written and approved, each specification formally defines a *baseline configuration* of the system or item. A baseline, in general, is an established and approved configuration, constituting an explicitly defined point of departure from which changes can be proposed, discussed, or accomplished.

Configuration control refers to the procedures by which changes to baselines are proposed and formally processed. These procedures involve standard classes and types of change proposals, as well as formal mechanisms for review, evaluation, approval, and authorization for implementing the proposed changes.

Configuration accounting refers to the reporting and documenting activities involved in keeping track of the status of configurations at all times during the life of a system. For production items of equipment, it includes the intricate process of maintaining the status of production changes, retrofits, and spare parts for all production items in the current inventory. In the case of a computer program item, it is principally a matter of maintaining and reporting the status of the specification, associated documents, and proposed changes.

Documentation and procedures

While the items of central reference in configuration management are contract end items, as distinct from data or services, the management process itself proves to be principally a matter of documentation and procedures. As indicated above, technical specifications are the principal substance of the Configuration Identification process. Configuration Control and Accounting are accomplished by means of other standard forms and reports. And, particularly in the case of complex computer program systems, account must also be taken of technical manuals and other data prepared for the using agency, whose contents are sensitive to changes in computer program configuration.

Hence, the major framework of configuration management and its sub-processes can be represented as a structure of the principal documents with which standard procedures are associated. This structure is illustrated in Figure 1, which shows (a) the specifications, as the baselines which are defined and managed, (b) the dependent procedural data, in the form of handbooks or manuals, and (c) the set of forms and reports which serve as tools for control and accounting. It may be noted that the computer program contract end item (CPCEI) is also represented, in the physical form of a tape. While the tape (or alternative form, such as punched cards) is not forgotten as the eventual object of these activities, most of our concern about working procedures is with the other elements. Additionally, it should be noted that events are related in a gross way to phases of the system life-cycle. In general, the structure begins at the outset of the Definition Phase with issuance of the System Specification, is expanded during the Acquisition Phase, and is subsequently maintained during the system's operational life. The specifications are established as the three baselines at successive times, and in dependent relations, during the developmental periods. However, an earlier baseline is not replaced by a new one; once established, all are maintained together, indefinitely.

Specifications

The *system specification* is typically written and issued by a procuring agency as the primary requirements document governing the developmental process for all contract end items. Its function is to define objectives and constraints for the system as a whole. It sets forth requirements for system performance and system testing, identifies all major parts to be developed or otherwise acquired, and allocates performance requirements to the parts. As a basis for information processing efforts, one of its important roles is to define and control essential functional interfaces among computer programs, computing equipment, communications links, and personnel. When issued at the outset of the Definition Phase it is established as the *program requirements baseline.* While it normally undergoes subsequent expansion and refinement, all changes must be approved by a central control board, documented, and coordinated with the agencies concerned.

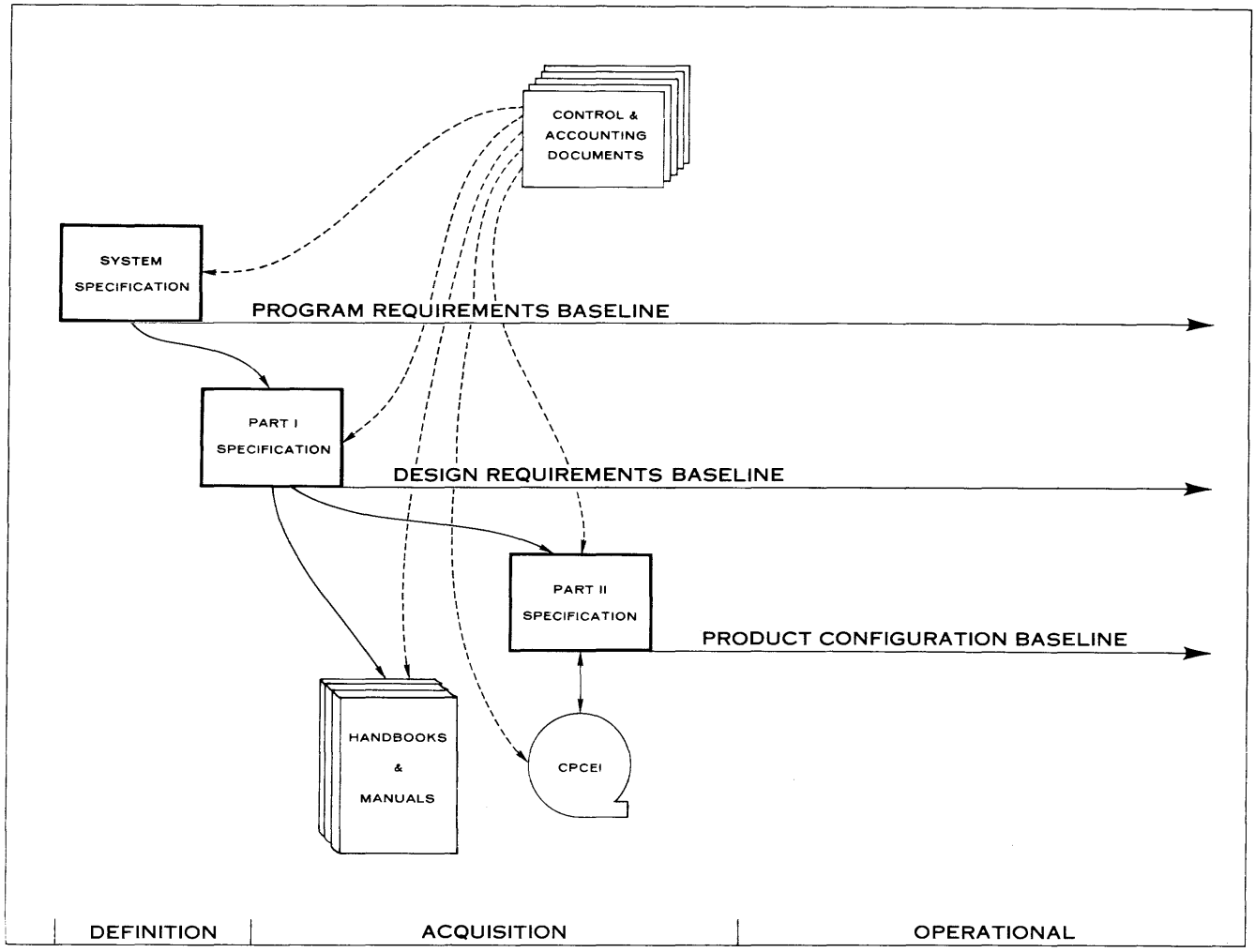


Figure 1—Sequence and structure of documents

The *Part I specification* for a computer program contract end item (CPCEI) is primarily a detailed compendium of information processing functions to be performed by the computer program. As a product of the Definition Phase analysis activities, it is not a computer program design document, as such, although it may specify design requirements or constraints and should reflect an appreciation of feasible computer program design solutions. It contains a detailed definition of all input, output, and processing functions to be accomplished, specifies all pertinent interfaces with equipment, personnel, and other computer programs, and identifies the means by which the eventual performance of specified functions will be verified by formal testing. The standard format which has been developed for the purpose[2] is written at a very general level to permit application to many types of computer programs, both large and small. As a specification, it does not contain planning information or procedural data pertaining to use or operation. Its functions are to govern the processes of computer program design, development, and testing, and to provide the primary instrument for management control throughout the course of a system program. Since it is written in mathematical, logical, and operational terms, rather than in technical computer programming language, it provides an essential vehicle by which the computer program functions can be communicated and managed at the level of operational performance. Part I Specifications for the collection of computer program and equipment elements for the system as a whole are verified for consistency with the System Specification and compatibility of interfaces, and are established collectively at the outset of Acquisition as the *design requirements baseline*. Because of its exceptional importance to the conduct of a system program, the Part I CPCEI Specification was chosen as a special topic to be amplified in another paper in this session.[3]

The *Part II specification* is a technical description of the CPCEI structure and functions. Essentially, it consists of elements which have appeared as computer program documentation in the past in a variety of forms and combinations—e.g., functional allocations, timing and sequencing, data base characteristics and organization, flow charts, etc. The Part II

Specification organizes these into a single, comprehensive description which includes the data base and listings. It is an "as-built" description, which the procuring agency accepts only after its completeness and accuracy have been formally verified, and normally after the computer program performance has been tested against performance requirements of the Part I Specification. Like the Part I, it does not contain operating instructions, test procedures, or other non-specification materials. Its subsequent technical uses are as an aid in correcting errors and designing changes. Following completion and verification, it constitutes the *product configuration baseline.* As in the case of preceding baselines, it is maintained subsequently through the processes of configuration control and accounting.

Procedural data

As indicated above, operating and instructional information is not contained in the computer program specification. This class of information is the proper subject of separate handbooks or manuals, which should be written specifically to meet the needs of operating and support personnel. Configuration management does not apply directly to such documents. However, since their content is typically sensitive to CPCEI performance and/or design characteristics, they must also be maintained to reflect approved changes to the CPCEI. Hence, they appear as prominent "impact items" in the configuration control processes.

Configuration control and accounting documents

Configuration control and accounting documents are standard forms and reports which serve as the instruments by which changes to established baselines are processed and recorded.

The *engineering change proposal* (ECP) is the vehicle by which a change is initiated. It is a form of long-standing use for equipment which has been adopted with appropriate modifications for computer programs. A completed ECP describes the nature and magnitude of a proposed change, estimated developmental requirements, and impact of the change on all associated documents and other system elements. During most of the Acquisition Phase, "computer program changes" normally refer to changes in the Part I Specification. At later stages, a fully implemented change will also involve the preparation of change pages to the Part II Specification and other derived documents, as well as incorporation of changed instructions into the computer program. Assuming that the Part II Specification is based upon completed design and testing of the altered computer program, changes are essentially implemented with the issuance of approved change pages to the specifications.

The *specification change log, end item configuration chart,* and *specification change notice* (SCN) are also standard forms, originally developed for equipment, which have been adopted with modified uses for computer programs. They constitute, respectively, (1) a cover sheet, (2) a listing of current pages, and (3) a summary of incorporated changes to accompany each issue of change pages to a specification. As a group, they are inserted into the revised specification, to provide ready indicators of the specification's current status.

The *version description document* is a collection of pertinent information which accompanies each new release of a computer program tape, or cards. Its purpose is to identify the elements delivered and record additional data relating to their status and usage.

The *configuration index* and *change status report* are items issued periodically (e.g., monthly) to inform interested agencies of the current status of the CPCEI configuration and proposed changes. The Index contains up-to-date listings of basic issues and revisions of the specifications and derived documents, with dates of issue and approved changes. The Change Status Report is a supplement to the Index which records the status of all current change proposals.

CONCLUSION

The preceding description has attempted to provide only an introductory overview of configuration management for computer programs. The documentation and procedures are the result of some years of study, experience, and coordination which were accomplished principally under the direction of the Technical Requirements and Standards Office at the Electronic Systems Division of Air Force Systems Command. At the time of writing this report (January 1967), the detailed requirements are in process of being inserted into the forthcoming revision of AFSCM 375-1,[4] which is scheduled to be issued within the next few months. A full set of requirements, in the form of a collection of the proposed changes to AFSCM 375-1, was also issued separately for interim use as ESD Exhibit EST-1 in May 1966.[2] Although that exhibit has had limited distribution, it is presently being used in a number of Air Force contracts and significant provisions have also been coordinated for joint use by NASA in managing computer programs for the Apollo program.[5] Experience to date has tended to confirm that the provisions are generally sound, and capable of meeting a num-

ber of long-standing needs. It is anticipated that refinements will be suggested by further use and study.

REFERENCES

1 M V RATYNSKI
The Air Force computer program acquisition concept
SJCC Proceedings 1967

2 ESD Exhibit EST-1:
Configuration management exhibit for computer programs
Electronic Systems Division Air Force Systems Command Laurence G Hanscom Field Massachusetts May 1966

3 B H LIEBOWITZ
The technical specification—key to management control of computer programming
SJCC Proceedings 1967

4 AFSCM 375-1:
Configuration management during definition and acquisition phases
Air Force Systems Command U S Air Force June 1 1964

5 B H LIEBOWITZ E B PARKER III C S SHERRERD
Procedures for management control of computer programming in apollo
TR-66-342-2 Bellcomm Inc September 28 1966

6 M S PILIGIAN J L POKORNEY
Air Force concepts for the technical control and design verification of computer programs
SJCC Proceedings 1967

The technical specification–key to management control of computer programming

by BURT H. LIEBOWITZ
Bellcomm, Incorporated
Washington, D. C.

INTRODUCTION

If one considers the life cycle of a system as described in the paper by Ratynski,[1] it becomes evident that effective management is dependent upon the presence of detailed specifications. In general, a specification describes what a product should be so that someone or some group can design and/or build it. In the previous paper the concept of a *two*-part specification was introduced–the first part describing the technical requirements for the item, the second part describing the actual configuration of the completed item. In this paper we will consider the content of the specification and its application to the management control of the computer programming process. Particular emphasis will be placed on part 1 of the spec. In doing so we will pose and answer two questions: (1) why is the part 1 spec so critical to management control? and (2) why, if so important, have so many programming efforts been characterized by its absence?

To answer the first question, let us consider the objective of and the steps required for management control.

The objective of management control is to assure that the resultant program does the job for which it was intended, is ready when needed, and is produced within expected costs. To achieve management control the managing group must: develop a standard which describes expected performance, evaluate actual performance against this standard, and take corrective action when undesired deviations from the standard occur. Performance in this context covers both the performance of the group developing the program and the performance of the program itself.

For computer programming, the cycle formed by these steps looks something like that shown in Figure 1. The standard is expected costs, estimated schedules, and desired technical characteristics. Evaluation is accomplished by means of reviews and tests. Corrective action is accomplished by allocations of resources, clarification of the standard and, when necessary, changes to the standard.

The part 1 specification is an important segment of the management standard. It documents the technical requirements for the computer program, thus providing both the basic "design to" guide for the programming contractor and the standard by which the user can evaluate the resultant program.* Without it the program developer must rely on word of mouth and informal notes and the user has no way of determining the validity of the evolving program. Thus there is no firm basis for management control.

To answer the second question, we must consider several factors, some common to both hardware and computer program development, others peculiar to computer program developments.

In the former category we find in some cases that lack of time prevents an adequate definition of requirements. Also, in some cases the nature of the effort is such that total requirements are not known prior to some experimentation with the system as (partially) built.

In computer programming we are faced with one other factor–the flexibility of the programmable digital computer. A somewhat exaggerated view of this capability had led unwary managers to the belief that, since changes to a finished program can be easily made, the time and costs to define requirements, and the need to exercise control once they have been

*For convenience in this paper, the managing group will be refered to as the "user" or "using agency"; the program developer will be referred to as the "contractor."

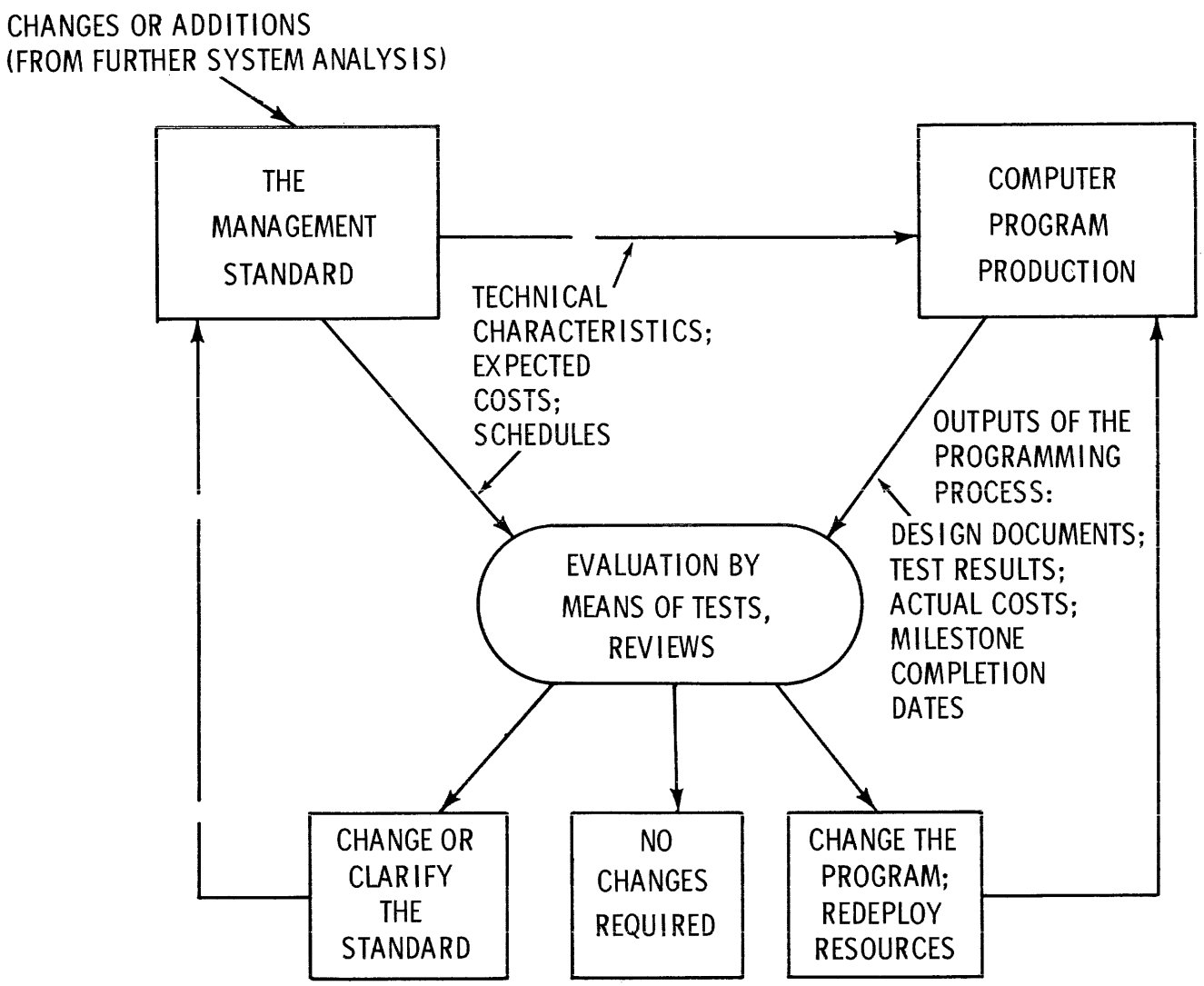


Figure 1 – The management control cycle

released, can be avoided in computer programming. Another difficulty is caused by the use of computers to provide flexibility in systems which operate in a changing environment. These systems are designed so that computer programs will absorb the major portion of expected changes. This allows firm definition of hardware requirements early in the development cycle of the system but makes it difficult to define the computer program requirements at the same time.

The impossibility under realistic conditions of establishing complete and definitive requirements in the programming process should not, however, be interpreted as a hopeless impasse to management control. On the contrary, if we accept as the general rule that the definition of requirements is an evolving process, we must conclude that a major function of management control is to insure that the development of the part 1 spec is an orderly one. This can be accomplished by providing mechanisms for the development of the spec and then for controlling changes to the spec as the requirements become better defined. This is actually what configuration management is all about as discussed in the paper by Searle and Neil.[2]

The rest of this paper will discuss the content of the first part of the specification, some methods for developing it in face of the realities mentioned above, and its relationship to the second part of the spec and other management control tools.

We begin with a brief description of what the first part of a computer program specification should contain.

The technical specification (Part 1)

Part 1 of the specification results from an analysis of the role the computer is to play in the system;

it is a translation of user needs to the language of computer systems analysts. The specification is developed in the definition phase prior to extensive design of the program.

Content of the specification

The computer program specification contains performance requirements, interface requirements, design requirements, and test requirements.

The performance requirements describe what the program is to do and the required data base characteristics. For each major function of the program the specification describes the type of data inputs, the required computer processing and the required outputs. The interface requirements describe the content and format of data transferred between the computer program and other computer programs and equipments. The design requirements delineate items, not directly related to the desired performance of the program, which constrain the design of the program, e.g., requirements for modularity and expandability. The test requirements define the types of tests needed to verify that the resultant computer program satisfies the performance and design requirements contained in the specification. Test requirements are included in the specification to allow the contractor sufficient time to adequately plan for running the required tests and to facilitate management control of the testing process.[3]

An outline of a model format for the first part of the specification is given in Figure 2. A description of a model spec is given in the Appendix of this paper. The format was developed with two things in mind: (1) to provide a convenient framework for the documentation of computer program requirements, and (2) to conform as much as possible to the format already in use for hardware specifications. The format and model were produced as part of an effort, undertaken on behalf of NASA's Apollo Program Office, to apply configuration management to computer programming.[4] This effort was closely coordinated with a similar one undertaken by the USAF's Electronic Systems Division.[5]

Development of the specification

Ideally, the user would produce a complete set of requirements prior to the design efforts of the contractor. With adaption of the phased system acquisition concept as discussed by Ratynski,[1] this ideal can be more closely approached than has been in the past. In this approach the interface, performance, design and test requirements would be developed in the definition phase prior to the contractor's design efforts in the acquisition phase (see Figure 3). However, in many cases for the reasons discussed earlier in this paper, some overlap of requirements analysis with design must be allowed if the program is to be available when it is needed.

An orderly development of requirements can still be achieved in these cases. The requirements can be built up in several ways depending on exact circumstances. In some cases the functions of what will eventually be a single program can be split into several subprograms, each of which can be developed separately. For example, in the case of an automated checkout system the final program may be composed of a supervisory system plus a package of test programs. The detailed requirements for the supervisory system may be relatively insensitive to the object under test and hence can be developed prior to the availability of the complete set of test program requirements. Significant design and development of the final program can then be accomplished prior to the total definition of computer program requirements.*

Another approach is to develop enough of the requirements to start design of the program at a high level, detailed design taking place as more requirements become available. For example, design of a program could commence subsequent to the establishment of a set of *qualitative* performance and design requirements. As more quantitative requirements – accuracies, response times, exact interface data formats, etc. – become known, the specification can be amended allowing the contractor to design the program in more detail. As requirements evolve, the program can be designed, evaluated (such evaluation perhaps aiding in the further definition of requirements) and flow charted. Coding, debugging, and assembly of the program would commence upon release of the completed requirements.

Even where the ultimate form of the system is not known at the inception of design – as in the case of some research and development projects – the requirements can be built up in an orderly fashion. To do this we borrow the concept of a prototype from hardware development. For many systems enough requirements can be developed in the definition phase to serve as the basis for producing a meaningful prototype computer program. This program can then be used in the operational system. As experience is gained by using the prototype, the requirements can be revised, reissued, and from them a "second gen-

*Another benefit of this approach is the isolation of those portions of the program which are expected to change; they in turn can be designed to be flexible, this being stated as a design requirement.

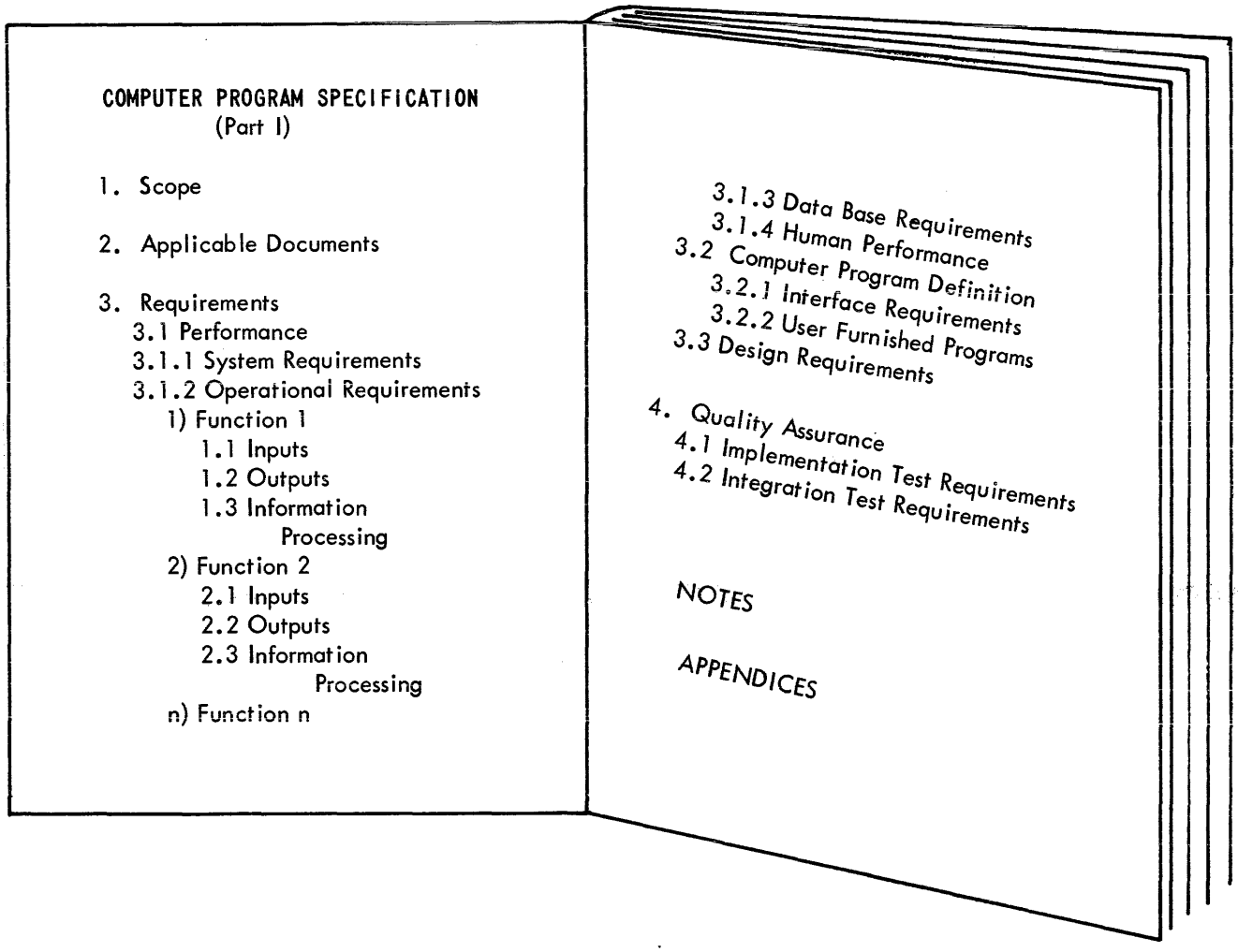


Figure 2—PART 1 of the computer program specification

eration" program developed. This iterative procedure can be repeated in controlled discrete steps until the final operational system is produced.

Variations and combinations of the above described approaches may be appropriate in particular efforts. Whatever approach is used, however, a common factor exists: as soon as sufficient requirements have been developed to allow initial design of the program, they are documented and officially released as the "design to" specification. This usually occurs when the performance and design requirements have been defined to a level of detail which permits the contractor to segment the program into component subprograms, allocate functions to the components, and develop flow charts to show the interrelationship among the components.

Control of the specification

Control of changes and additions to the part 1 specification is essential if it is to be used as the standard for controlling the design and development of the computer program.

Formal control, by the user, begins with the release of (some subset of) the requirements as the part 1 specification. Control is vested in a change control board (CCB) made up of user personnel who are aware of the technical effects of changes throughout the system. Any change to the specification, except for correction of editorial errors, must be approved

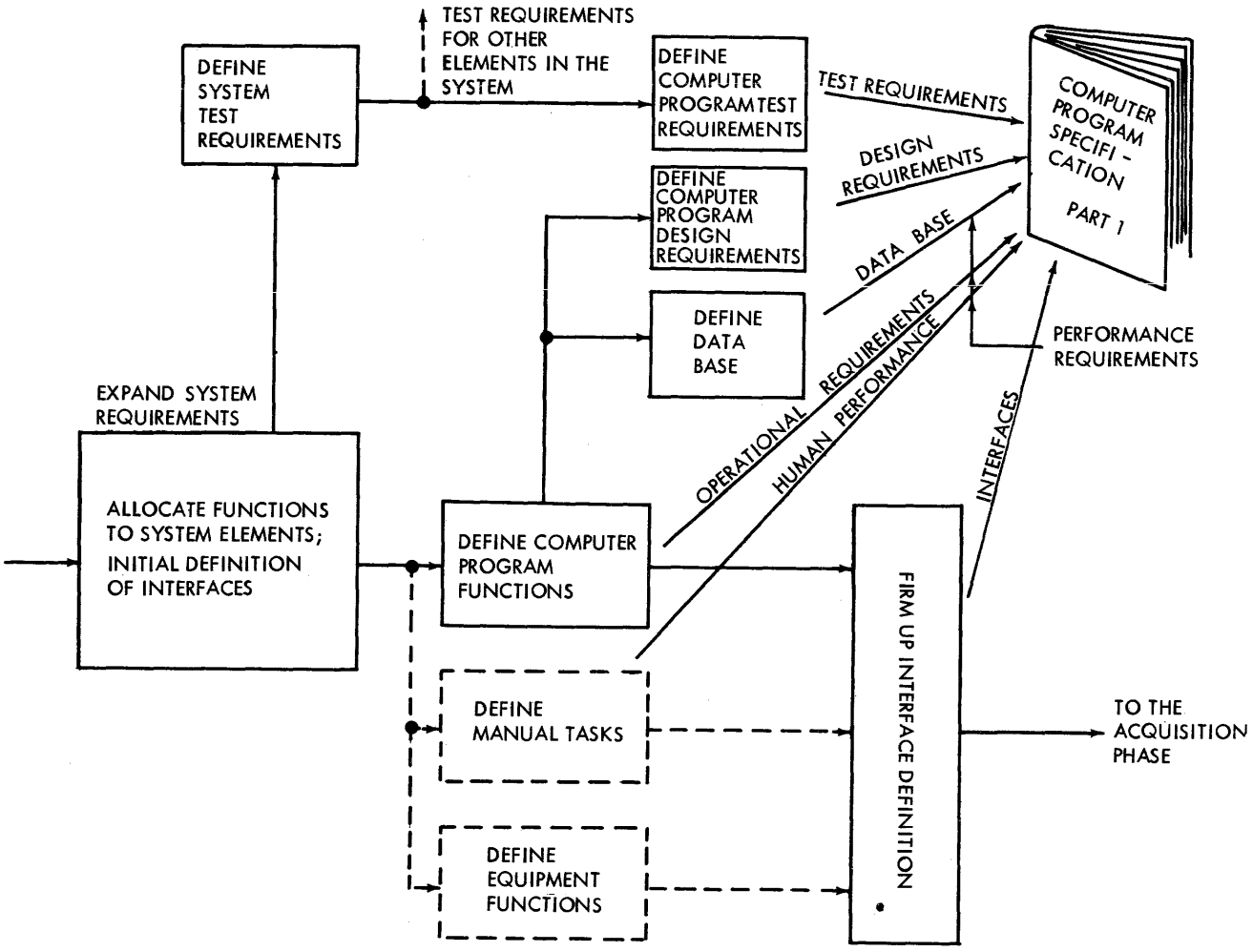


Figure 3 – Definition phase activities leading to the development of the Part 1 specification

by the CCB before the change can be incorporated into the specification. All approved changes must be documented and disseminated to all agencies affected by the changes.†

The CCB must be able to recognize and react to different kinds of changes to the part 1 specification. Many of the early changes to the specification will actually be additions, filling in areas not previously defined. Changes of this type will come from the group in the using agency that developed the original requirements. Changes will also come from other groups in the using agency as a result of further definition of or changes to the system's requirements. The CCB transmits and coordinates these types of changes to the contractor. Other proposed changes may come from the contractor as he uncovers inconsistencies in the requirements, or formulates new concepts as to what the program should do. If acceptable to the CCB, they are incorporated into the specification.

†A discussion of change control and accounting documents which aid in the change control process is given in the paper by Searle and Neil.[2]

The specification as a management control tool

Once released, the part 1 specification serves as a valuable tool in controlling the contractor's design; in assuring the quality of the program, and in accepting the resultant computer program. In many cases it is also a valuable tool in procuring the services of a programming contractor.

Procurement

In those cases where the services of an outside programming contractor are required, the part 1 specification, included as part of a statement of work[6] in a request for proposal, helps define the task to be done. The more complete the specification the better

the chances are of getting realistic bids from contractors.

Design control

The specification provides the standard by which the contractor's design may be evaluated. This can be done in several reviews during the programming process, permitting detection of design errors (or faults in the requirements analysis) early enough to do something about them. At least two types of reviews are helpful: a preliminary design review (PDR), and a set of critical design reviews (CDR's).[3]

The PDR is held early in the design process. The contractor's initial overall design approach is reviewed with respect to the part 1 specification. The review gives the user an opportunity to evaluate the proposed design to see if it will meet the requirements as stated in the spec. After the PDR, the contractor proceeds to the detailed design of the individual components of the program.

Once the detailed design of an individual component is complete, it is reviewed in a CDR, prior to coding. This provides an additional check on the ability of the computer program to meet the design and performance requirements. Since the detailed design of a large scale computer program may take considerable time, several CDR's may be held.

Quality assurance

The "quality" of a computer program is determined by its ability to do the job for which it is intended, to be modified as required, to be maintained, and to be understood by others than those who produced it. Although quality in this sense is difficult to measure, it can be enhanced by the availability of programming standards, the application of good documentation techniques, and comprehensive testing of the evolving program. The total specification—both part 1 and part 2—can be used as an instrument in achieving these conditions.

Programming standards are used to control and guide the efforts of programmers during the acquisition phase to ensure that a self consistent computer program is developed (see Figure 4). They are usually of three types: standards of formats, for example, for punched cards and flow charts; standards of design techniques; and standards of procedures regulating programming personnel.[7] Although the responsibility for developing the standards usually lies with the programming contractor, the requirement for producing them is levied by the user in the design requirements section of the part 1 specification. The user may also include detailed standards in part 1 if appropriate. For example, if the user knows the program will have to be modified extensively in operational use, he could include a requirement in the part 1 specification limiting program module size to 1,000 storage locations.

The part 1 specification also provides a vehicle for indicating the types of tests the contractor must perform during the programming process. These tests are used to determine if the program components are error free, a condition which must be approached if a high quality program is to be produced.

Another essential aspect of quality assurance is documentation of the resultant program. Without adequate documentation, the program cannot be modified or corrected, causing it to lose its value as time progresses. Part 2 of the specification provides a description of the program and as such is the instrument for making and controlling changes to the program in the operation phase. The part 2 specification is a collection of information that is generated during the acquisition phase of any well managed programming effort (see Figure 4). It contains:

(1) a description of the overall program design including high level flow charts;
(2) programming specifications for the individual components of the computer program;
(3) the detailed flow charts for the components;
(4) and program listings.

A complete description of a suggested format and content of the part 2 spec is given in the aforementioned government manuals.[4,5]

Acceptance of the program

The role of the part 1 specification in the formal acceptance of the program is discussed elsewhere in this session. Suffice it to say here that part 1 has a twofold role in the acceptance process. It provides the quantitative and qualitative measures by which the performance of the program can be evaluated; it also provides specific requirements for the types of tests and inspections necessary to make this evaluation.

CONCLUDING REMARKS

Computer program requirements, documented in the part 1 specification, provide a standard by which the performance of the contractor and the performance of the resultant computer program can be evaluated.

Despite some inherent problems which may prevent a user from developing complete requirements prior to the design of a computer program, it is possible to use the evolving requirements as a key management tool during the program's development.

An initial part 1 specification can be released early in the programming process for use by a contractor as the basis for the design of the program. The remaining requirements can be transmitted to the contractor in an orderly fashion as they are developed. All sub-

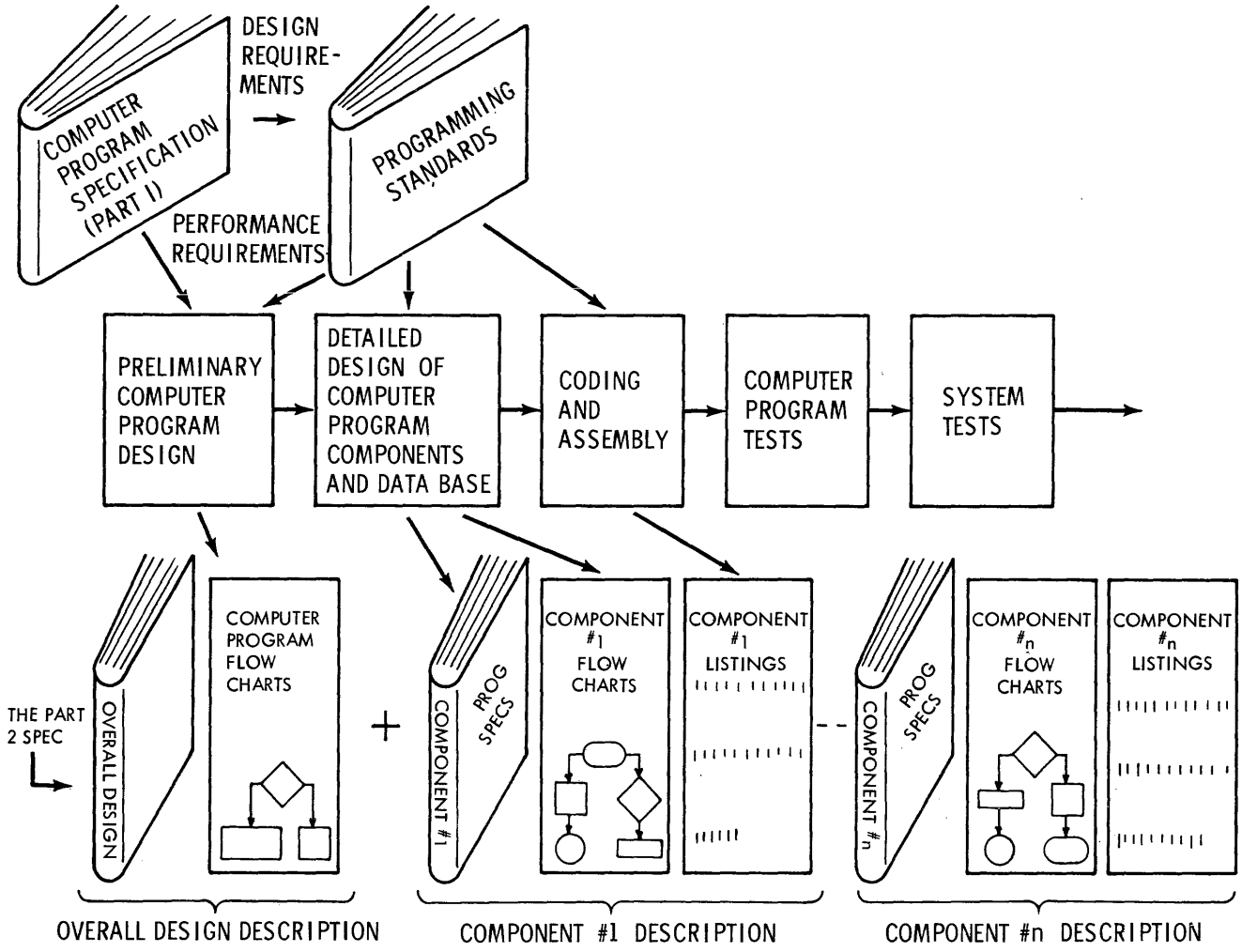


Figure 4 – Programming standards and the Part 2 specification in the acquisition phase

sequent changes to part 1 can be controlled by a change control board, composed of user personnel, to insure the availability of up-to-date requirements consistent with the requirements of any interfacing elements.

Several benefits accrue from having an up-to-date part 1 specification throughout the development cycle of the program. The user can evaluate the contractor's design at periodic intervals. The contractor is provided a basis by which he can develop programming standards, which along with the program description documented in part 2 of the specification is an essential tool in assuring the quality of the program. Also, the user is provided with a measure for determining the acceptability of the program since the result of formal acceptance tests can be compared against the performance requirements in the part 1 specification.

APPENDIX - A model for the Part 1 specification

This appendix presents a more detailed description of the types of information that should be included in a computer program requirements specification. The format illustrated in Figure 2 will be used as the basis for discussion.

Performance requirements (3.1)

The performance requirements put bounds on the expected performance of the program. This is done at several levels. The first level specifies the system characteristics which define the environment that the program operates in (3.1.1). For example, in a space tracking system this could be the number and types of radars involved, the maximum number of space vehicles to be tracked, the numbers and types of displays to be driven, etc. For a payroll program this could include such things as the number of employees, the types of deductions, the salary ranges, etc.

The second level provides an identification of the distinct functions that the computer program must perform. The functions are described, with the aid of a functional block diagram, under the heading "operational requirements" (3.1.2). The block diagram illustrates the relationships of the functions to each other to clarify the textual descriptions that follow in the spec.

The third level provides the inputs, outputs, and required processing for each function depicted in the block diagram. In the space tracking example, as a case in point, the first identified function might be that of orbit determination. A lead paragraph (3.1.-2.1) describes the function in general terms with particular emphasis on its relationship to other functions.

Then the source and type of inputs associated with the function are described (3.1.2.1.1). In our example this might be spacecraft position data from each tracking station and a table of past spacecraft positions generated by the computer program. For each of the input sources, such things as data rates, accuracies, units, and limits should be given.

The types and destinations of data outputs associated with the function are then given (3.1.2.1.2). For example, one such output might be an orbital prediction used for driving a flight controller's display. Again such things as data rates, accuracies, units and limits are specified.

The information processing required for the function is then described (3.1.2.1.3). This is done by prose and mathematical descriptions including necessary logical concepts, timing requirements, formulas and processing accuracies. In our example this might include among other things, the differential equation describing the orbit, the accuracy with which the equation must be solved, and the interval of time within which it must be solved.

The next major class of requirements considered in the performance section of the spec is the requirements for the data base – the collection of data that the program operates on (3.1.3). For many programs the storage requirements for data may exceed the requirements for instructions and must be carefully considered before the program is written. This consideration should include such things as the volume of data, a definition of data classes, for each datum, a description of the units of measure and ranges. If special methods are required to convert the data to a form suitable for use by the computer program, they should be specified. If the program is to be used at several sites where the data values will vary from site to site, the site adaption procedures should be specified.

Many computer programs have extensive human interfaces and like hardware items should be designed with human performance in mind. Therefore, the part 1 specification includes such things as clarity requirements for displays, times for human decision making, maximum display densities, etc. (3.1.4).

Computer program definition (3.2)

While the performance requirements may take the bulk of analysis time and occupy most of the content of the specification, there are other important factors that must be considered prior to the design of a computer program. Two of these factors – the interfaces of the computer program with the other elements of the system, and the available computer programs that can or must be used as components of the desired program – are specified under the category, "computer program definition."

The interfaces with other computer programs, communication devices, and equipments must be specified prior to detailed design of the computer program (3.2.1). This includes all relevant characteristics of the computer or class of computers the program is to operate in, including such things as available memory, programming languages, word size, etc. The relationship of the computer program to other equipments and programs should be portrayed in a block diagram, with detailed paragraphs in the spec describing the individual interfaces. The descriptions should include such things as data formats, data rates, and data content. In addition, this portion of the spec should describe all requirements imposed upon *other* system elements as a result of the design of the program. For example, certain program requirements may affect the design of a display console; if so, they should be documented in the spec.

In many programming efforts there may be available already completed programs which are of potential use to the program designers. For example, a tightly coded numerical routine for solving differential equations may exist in a program library. If there are such programs, the requirements to use them should be stated in the spec (3.2.2).

Design requirements (3.3)

There are certain requirements which affect the design of a computer program, which are distinguishable from the performance requirements. These requirements result from general considerations of usability and maintainability, and may include such things as requirements for: the use of programming standards; program organization; special features to facilitate the program's testing; and expandability. These types of requirements should be considered in the definition phase and included in the spec.

Quality assurance (4.0)

One of the major functions of the specification is to define the tests and demonstrations necessary to qualify the program for operational use. For most computer programs the time and resources expended for testing exceed those for any other activity in the programming process. Unless the requirements for testing are considered early enough, there may be delays later in the process when these needs arise. Also, in a large system the requirements for the testing of a computer program are directly related to the requirements for testing the entire system. For these reasons test requirements should be considered in the definition phase and included in the specification.

A method of verification should be given for each of the performance and design requirements stated in the specification. The requirements should be stated to a level of detail which will permit detailed test planning by the program developer. The specification itself should not be a test plan. The types of tests that should be considered include implementation tests, integration tests, preliminary qualification tests, final qualification tests, string tests, etc.[3]

REFERENCES

1 M V RATYNSKI
The Air Force Computer Program Acquisition Concept
SJCC Proceedings 1967

2 L V SEARLE and G NEIL
Configuration Management of Computer Programs by the Air Force Principles and Documentation
SJCC Proceedings 1967

3 M S PILIGIAN and J L POKORNEY
Air Force Concepts for the Technical Control and Design Verification of Computer Programs
SJCC Proceedings 1967

4 NPC 500-1
Apollo Configuration Management Manual (Exhibit XVIII)
National Aeronautics and Space Administration May 18 1964

5 ESD EXHIBIT EST-1
Configuration Management Exhibit for Computer Programs
Electronic Systems Division Air Force Systems Command Laurence G Hanscom Field Bedford Massachusetts May 1966

6 B H LIEBOWITZ E B PARKER III and C S SHERRERD
Procedures for Management Control of Computer Programming in Apollo Appendix I
TR-66-320-2 Bellcomm Inc September 28 1966

7 IBID APPENDIX VI

Air Force concepts for the technical control and design verification of computer programs

by M. S. PILIGIAN and J. L. POKORNEY, *Captain, USAF*

Electronic Systems Division
L. G. Hanscom Field
Bedford, Massachusetts

INTRODUCTION

While much has been written about the design and development of large scale computer-based systems, little has been published about the testing of these systems and in particular about the testing of computer programs within the context of a system. Similarly, while extensive techniques for design control of system hardware have been developed by the Air Force over the past five years, technical control and design verification procedures for computer programs have not been aggressively investigated. An Air Force project was established at the Electronic Systems Division (AF Systems Command) to rectify this situation. Concepts resulting from this investigation are presented and current procedures to insure design integrity of computer programs by technical reviews of the designer's efforts and by tests during program development are summarized.

Uniform specifications

Fundamental to Air Force management of computer program design, development and testing is the definition of computer programs as deliverable contract end items and the adaptation of the Air Force uniform specification program to these end items.[1,2] The uniform specifications,[3] both at the system and computer program contract end item (CPCEI) level, consist of two basic sections; Section 3, performance/design requirements section, and Section 4, the quality assurance or test requirements section. It should be realized that even as early in the design process as the preparation of the system and end item specifications, the methods of testing end item performance against technical requirements should be known. It is a waste of time and money to specify technical requirements if a method of performance verification is not available to evaluate the computer program once it is developed. Generally, a one to one relationship exists between Sections 3 and 4 of the specification. Thus each requirement of Section 3 will have an appropriate test requirement and test method identified in Section 4. The specified requirements form the basis for three formal technical reviews through the system and contract end item design and development. Concurrently the specification test requirements are the basis for subsequent test planning documentation and system testing at both the contract end item and system performance levels. The relationship between the specifications, design reviews and test program is illustrated in Figure 1. The lower blocks of the figure identify the design reviews during the system life cycle each of which will now be discussed in more detail.

System design review

The System Design Review (SDR) is held late in the Definition Phase* of the system life cycle.[4] The purpose of this first review is to study the contractor's system design approach. At the SDR a critical examination is performed to insure that contractor's design reflects a proper understanding of all technical requirements. An analysis of contractor documentation in the form of functional diagrams, trade study reports, schematic diagrams, initial design specifications, etc. is conducted. A prime objective of the SDR is to review the allocation of functional requirements to the various system segments and contract end items. Thus, for computer programs, the SDR must insure that only those system requirements that can be realistically satisfied by

*Phases as discussed here (i.e. Conceptual, Definition, Acquisition and Operational Phases) refer to the 4 phases of the System Life Cycle as defined in AFSCM 375-4, System Program Management Procedures.

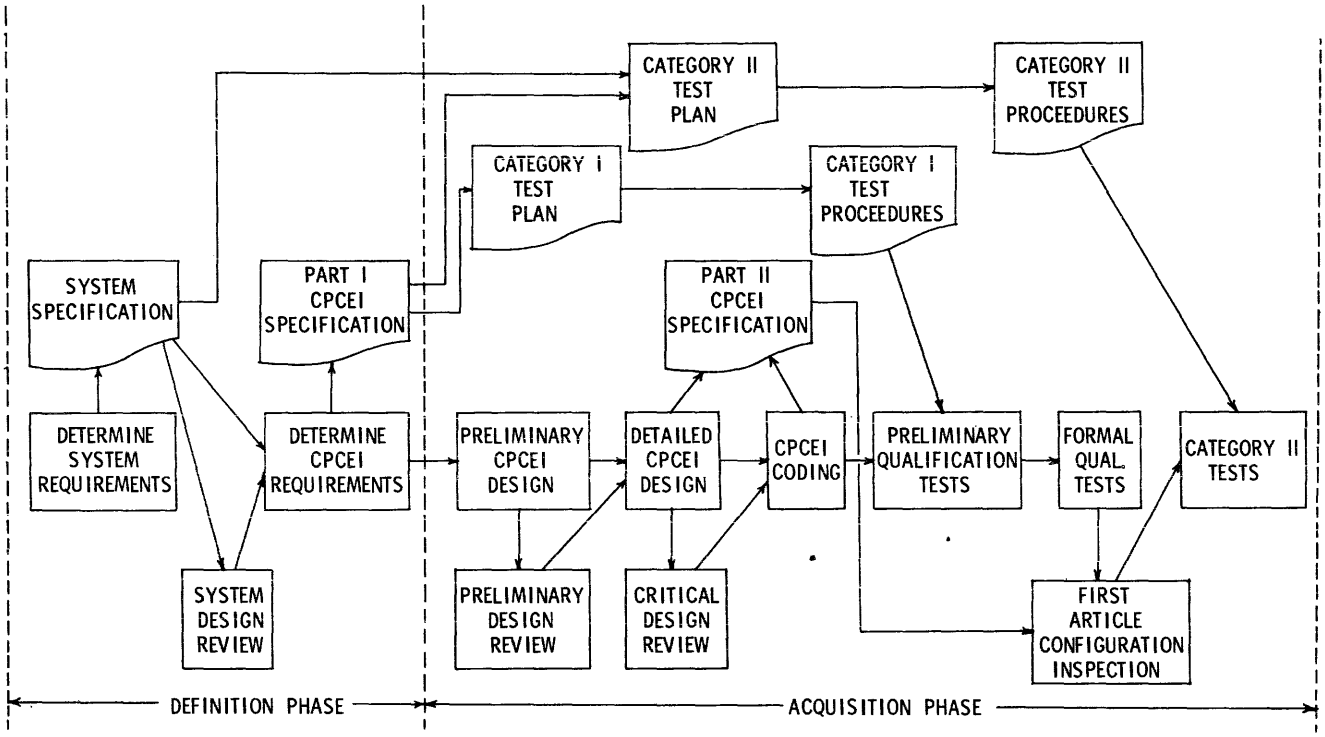


Figure 1 – Computer program design flow

computer programs have been allocated to computer program contract end items (i.e. operational, utility, diagnostic, etc.). Prior to the conduct of the SDR, trade-off studies concerning equipments vs. computer programs must have been completed to provide a cost effective allocation of requirements. Satisfactory completion of the SDR permits preparation of the Part I specifications ("design to" specifications) for all CPCEI's. These specifications form the basis for the second technical review in the design process.

Preliminary design review

The Preliminary Design Review (PDR) is normally held within 60 days after the start of the Acquisition Phase. Concurrently the preliminary design of the CPCEI can progress based upon the approved "design to" specifications for the end item. The purpose of the PDR is to evaluate the design approach for the end item or group of end items in light of the overall system requirements; thus, the prime objective of the PDR is achieving design integrity. A review of the interfaces affecting the computer program contract end item is an important element of a PDR. Emphasis is placed on verification of detailed interfaces with equipment and with other CPCEI's. At the PDR the instruction set of the computer to be used must be firmly established. The programming features of the computer, e.g. interrupts, multiprocessing, time sharing, etc. must be known. All external data formats and timing constraints must be identified. The computer program storage requirements and data base design are reviewed for technical adequacy at this time. The structure of the computer program contract end item is also reviewed at the PDR. During the initial design process for a complex CPCEI, the requirements of the Part I specification which are function-oriented are allocated to computer program components or modules. The relationship of the components of a typical CPCEI to the functions identified in a Part I CPCEI specification is shown in Figure 2. The allocation of functions to computer program components within the CPCEI is examined at the PDR. The primary product of the review at this level is establishing the integrity of the design approach, verifying compatibility of the design approach with the Part I specification, and verifying the functional interfaces with other contract end items in order that detailed design of the CPCEI and its components can commence.

(N) FUNCTIONS	ATR	BAG	MIM	RST
RADAR INPUTS				X
TRACKING	X	X		X
IDENTIFICATION		X	X	
DISPLAY CONTROL		X		
DATA TRANSFER	X			X

(M) COMPUTER PROGRAM COMPONENTS

Figure 2 – Allocation of performance functions to computer program components

Critical design review

The Critical Design Review (CDR) is a formal technical review of the design of the computer program contract end item at the detailed flowchart level. It is accomplished to establish the integrity of the computer program design prior to coding and testing. This does not preclude any coding prior to the CDR required to demonstrate design integrity, such as testing of algorithms. In the case of a complex CPCEI, as the design of each component proceeds to the detailed flowchart level, a CDR is held for that component. In this manner, the CDR is performed incrementally by computer program components, and the reviews are scheduled to optimize the efficiency of the overall CDR for the end item as a whole. Due to the varying complexity of the parallel design efforts for CPCEI components, it would be unreasonable to delay all of the components being developed to hold one CDR for the computer program end item.

At the CDR, the completed sections of the Part II CPCEI specification (detailed technical description) are reviewed along with supporting analytical data, test data, etc. The compatibility of the CPCEI design with the requirements of the Part I specification is established at the CDR. "Inter" interfaces with other CPCEI's and "intra" interfaces between computer program components are examined. Design integrity is established by review of analytical and test data, in the form of logic designs, algorithms, storage allocations and associated methodology. In general, the primary product of the CDR is the establishment of the design as the basis for continuation of the computer program development cycle. Immediately following the CDR, coding of individual components takes place and the process of checkout and testing of the components begins.

Computer program testing

System testing as defined by the Air Force is divided into three classes or categories of testing, two of which, Category I and II[5] are important in development testing of Air Force systems and will be discussed here. Category I tests for CPCEI's are conducted by the contractor with active Air Force participation. These activities, when properly planned and managed, will normally proceed in such a way that testing and functional demonstrations of selected functions or individual computer program components can begin early during acquisition and progress through successively higher levels of assembly to the point at which the complete computer program CEI is subjected to formal qualification testing. Since the total process is typically lengthy and represents the major expense of computer program acquisition for the system, the test program includes preliminary qualification tests at appropriate stages for formal review by the Air Force. While the tests are preliminary in nature (they do not imply acceptance, or formal qualification), they do serve the necessary purpose of providing check points for monitoring the contractor's progress towards meeting design objectives and of verifying detailed performance characteristics, which, because of sheer numbers and complexity, may not be feasible to verify in their entirety during formal qualification testing. Category II tests are complete system tests, including the qualified computer program end items, conducted by the Air Force with contractor support in as near an operational configuration as is practicable.

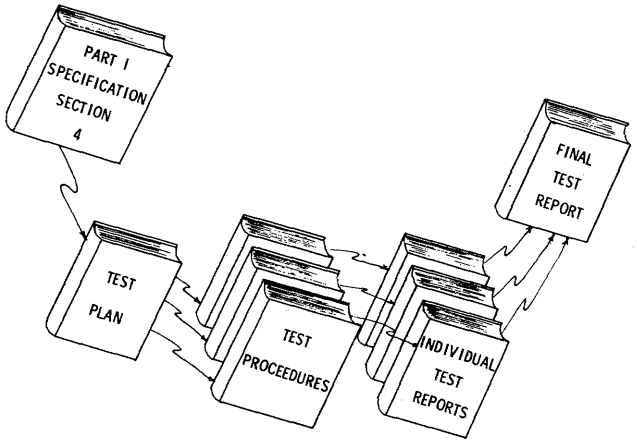


Figure 3 – Test documentation

Computer program testing is accomplished in accordance with Air Force approved test documentation as illustrated in Figure 3. As previously discussed, test requirements and corresponding test methods (to the level of detail necessary to clearly establish the scope and accuracy of the methods) are contained in Section 3 and Section 4 of the CEI specification. The Category I test requirements are further amplified in the contractor-prepared Category I Test Plan. This document contains comprehensive planning information for qualification tests; complete with schedules, test methods and criteria, identification. The Category I test requirements are further amplified in the contractor-prepared Category I computer programs and personnel. The Test Plan forms the basis for Category I Test Procedures which are also prepared by the contractor. These describe individual Category I qualification tests in detailed terms, specifying objectives, inputs, events, recording/data reduction requirements and expected results. Actual test results are reported in a formal Category I Test Plan. This document contains comprehensive planning information for qualification tests; complete with schedules, test methods and criteria, identification of simulated vs. live imputs and support require-

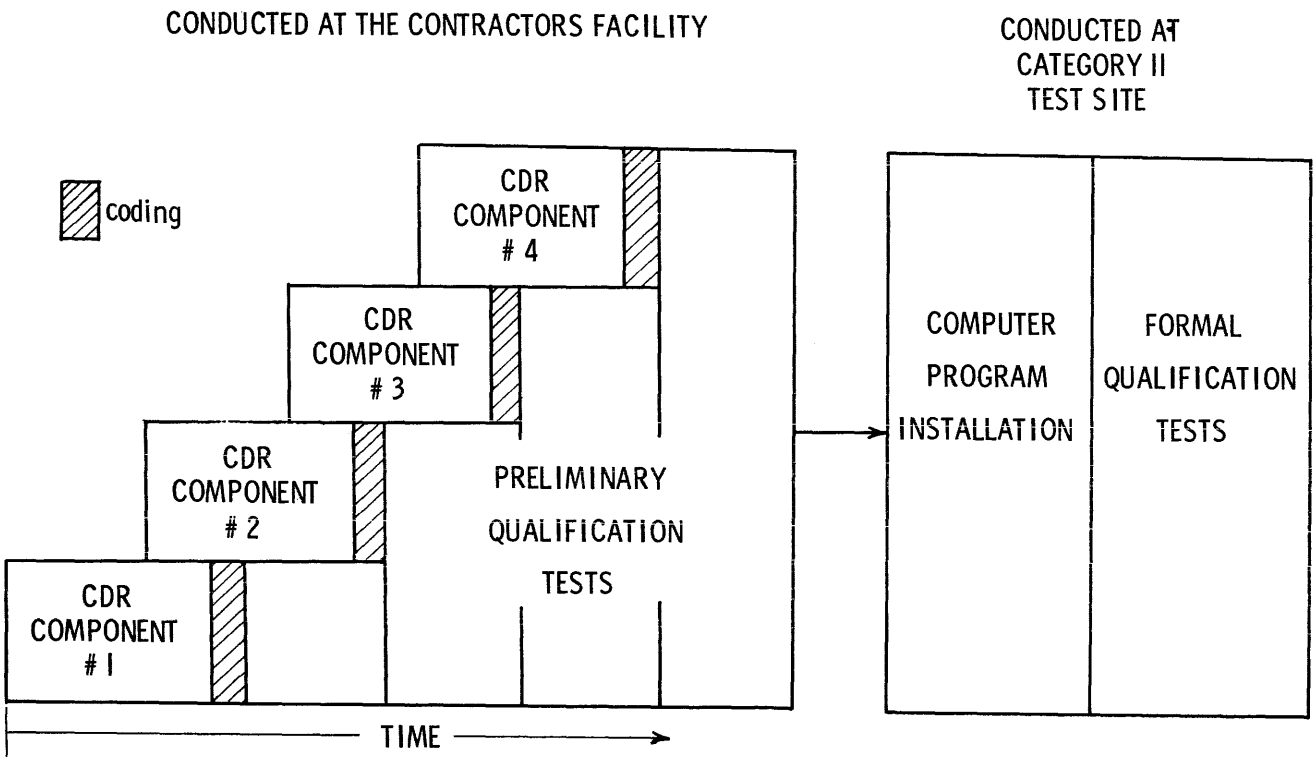


Figure 4 – Computer program contract end item critical design review and Category I tests

ments for test equipment, facilities, special test computer programs and pronnel. The Test Plan forms the basis for Category I Test Procedures which are also prepared by the contractor. These describe individual Category I qualification tests in detailed terms, specifying objectives, inputs, events, recording/data reduction requirements and expected results. Actu test results are reported in a formal Category I Test Report.

Category I (qualification) testing of computer programs

The Category I test program verifies that the computer program contract end item satisfies the performance/design requirements of the Part I "design to" CPCEI specification. The test program must be designed to insure that all of the functional requirements, as translated into computer program components, are tested and that requirements are not lost in the translation. The program is divided into two major classes of tests: Preliminary Qualification Tests (PQT) and Formal Qualification Tests (FQT). The former are designed to verify the performance of individual components prior to an integrated formal qualification of the complete CPCEI.

Preliminary qualification testing

The PQT phase is conducted incrementally by components in the same manner as the Critical Design Review. Figure 4 depicts the relationship between CDR and the Category I test program. The crosshatched blocks in Figure 4 indicate coding of individual computer program components. The Preliminary Qualification Tests are modular and a "building block" effect occurs as testing progresses. As each computer program component is added and each PQT conducted, increased confidence develops in the CPCEI being tested. Generally, parameter tests are conducted prior to and in parallel with Preliminary Qualification Tests.[6]

Parameter tests are those designed to prove that an individual sub program satisfies the detailed design specification, not that the program performs as coded. These tests compare the actual operation of each sub program against the design specification. Parameter testing usually requires a utility system incorporating sophisticated parameter test tools. These test tools, which are computer programs themselves, allow more efficient testing because they increase the ease with which a test can be specified, implemented and analyzed. They allow a programmer to easily input data, make corrections and record results. In addition to the test tools, the programmer needs the compiled or assembled program, the Part I specifications, the test plan and the test procedures.

In parameter tests, the programmer must input a simulated program environment to the computer. The environment should include the broadest range

of anticipated inputs, including some illegalities. The program is operated in this simulated environment, and the actual outputs are compared with the expected outputs. After each test run, the programmer analyzes the results and makes corrections to the code. All corrections are verified by submitting them to a parameter test once again.

Assembly testing verifies that the computer program components in the contract end item interact according to design specifications. It is conducted with simulated inputs in order to minimize the effects of people and equipment and allows a broad range of input conditions to be simulated. Elaborate test tools, such as input simulation, recording, and reduction programs, are required to conduct assembly tests. Since these programs take time to prepare, test requirements such as instrumentation must be anticipated. For example, provision must be made for core storage to accommodate test control and test recording programs along with the program being tested.

At the conclusion of Preliminary Qualification Testing, all of the computer program components will have been integrated and tested and the CPCEI is being ready for formal qualification and acceptance.

Formal qualification testing

Qualification testing of a complex computer program contract end item requires extensive use of simulation techniques. The use of these techniques is dictated by the high cost of providing overhead computer facilities or by the unavailability of new computers undergoing a parallel design and development effort. Although Preliminary Qualification Tests will make maximum use of simulation techniques, the Formal Qualification Tests of an operational CPCEI will require live inputs, live outputs and operationally-configured equipment. A prerequisite, then, of FQT is usually the installation and checkout of the CPCEI in an operationally-configured system at the Category II test site. The exception would be in the case of a support CPCEI such as a compiler that would require live inputs, e.g. radar data, and could be fully qualified at the contractor's facility. To provide reliable data during FQT, the CPCEI installation requires fully installed and checked out equipment CEI's. The first opportunity for FQT will normally occur at the Category II test site after equipment CEI's that have successfully passed First Article Configuration Inspection have been installed and checked out and an operationally-configured system exists. FQT is conducted subsequent to installation and checkout of the CPCEI. The conclusion of FQT signals the end of the Category I test program. The CPCEI will have been fully qualified and all of the requirements of the Part I specification should have been satisfied except for those requirements of the Part I specification that can only be demonstrated during a Category II system test. After successfully passing this phase of testing, the CPCEI is fully integrated into the system and is ready for system testing.

First article configuration inspection

With CPCEI design and testing essentially completed, Part II of the CPCEI Specification is available for review. The Part II specification provides a complete and detailed technical description of the CPCEI "as built," including all changes resulting from prior testing. It will accompany the CPCEI to each installation or site and functions as the primary document for "maintenance" of the CPCEI. Consequently, the technical accuracy and completeness of the Part II specification must be determined prior to its acceptance by the Air Force. The First Article Configuration Inspection (FACI) provides the vehicle for the required review; thus it is an audit of the Part II CPCEI specification and the CPCEI as delivered. The primary product of the FACI is the formal acceptance by the Air Force, of (1) the CPCEI specification (Part II) as an audited and approved document, and (2) the first unit of the computer program contract end item. Air Force acceptance of the CPCEI is based on the successful completion of the Category I Test Program and the FACI, but it does not relieve the contractor from meeting the requirements in the system specification. Subsequent to FACI, the configuration[7] of the CPCEI is essentially controlled at the machine instruction level so that the exact configuration of the CPCEI is available for Category II system testing.

Category II system testing

After acceptance of the CPCEI, the Air Force conducts an extensive Category II system test program with the objective of demonstrating that the total system meets the performance/design requirements specified in the System Specification. Insofar as the computer programs are concerned, Category II testing will verify the CPCEI's compatibility with the system elements and its integrated performance in meeting system requirements in the live environment, with operational communications, personnel, etc. Residual design and coding errors discovered in this phase of testing are corrected prior to the system becoming operational.

SUMMARY

The techniques for design reviews and testing presented in this paper provide a means of insuring the

design integrity of computer programs during the lengthy design and development cycle. It provides the Air Force with technical control, at discrete phase points, which was not before available. To provide this control, existing Air Force Systems Command management techniques were assessed and adapted to computer programs. No attempt has been made to equate computer programs with equipments; rather, the requirement for similar technical controls has been recognized with due consideration for the inherent differences between computer programs and equipment. While these techniques were developed for computer programs within the context of large computer-based systems, they are and have been readily adaptable to small individual computer program procurements. More detailed information on requirements and procedures are included in ESD Exhibit EST-1 and ESD Exhibit EST-2, published by the Electronic Systems Division.

Though the above techniques have been used on contracts at ESD, none of the programs have progressed through the complete cycle. The limited experience to date indicates that the techniques are feasible, they do provide vitally needed Air Force technical control and visibility and in turn they have been useful to the contractors as a formal management scheme and a means for mutual understanding and problem resolution.

REFERENCES

1 ESD Exhibit EST-1
Configuration management exhibit for computer programs
Test Division of the Technical Requirements and Standards Office Electronic Systems Division Air Force Systems Command L G Hanscom Field
Bedford massachusetts May 1966

2 AFSCM 375-1
Configuration management during definition and acquisition phases
Andrews AFB Washington D C
Headquarters Air Force Systems Command June 1 1964

3 B H LIEBOWITZ
The technical specification - key to management control of computer programming
SJCC Proceedings 1967

4 M V RATYNSKI
The Air Force computer program acquisition concept
SJCC Proceedings 1967

5 AFR 80-14
Testing/evaluation of systems subsystems and equipments
Washington D C
Department of the Air Force August 14 1963

6 LEONARD A FARR
A description of the computer program implementation process
System Development Corporation
TM-1021/002/00 February 25 1963

7 L V SEARLE G NEIL
Configuration management of computer programs by the Air Force: principles and documentation
SJCC Proceedings 1967

Univac ® 1108 multiprocessor system

by D. C. STANGA

Univac Division, Sperry Rand Corp.
Roseville, Minnesota

INTRODUCTION

Two prime objectives existed during initial conception of the 1108 Multiprocessor System. These were:

- Increased system performance, and
- Increased system availability.

Increased performance in a "central system" was achieved via multiple processing units, multiple input/output units, and multiple access paths to critical peripheral devices. To accommodate this increase in computer capability the main memory was increased in size to provide storage for a large number of resident programs.

Increased availability was achieved by providing multiple access paths to all system components, and by providing for removal of system components for offline testing and maintenance.

In order to accomplish the prime objectives, several self-imposed restraints were placed on the design considerations, these are: modifications to the 1108 Processor would be minimal, the two processors (1108 & 1108 II) would be functionally identical, all peripheral subsystems and existing memory modules must operate in the new configuration. The self imposed restraints were intended to minimize development time and ensure that the existing upward compatibility between 1107 and 1108 Processors would be continued through the 1108 II Processor.

System configuration (hardware)

Figure 1 shows a maximum 1108 Multiprocessor System configuration, with the exception of the Peripheral Subsystems which are essentially unlimited with respect to the rest of the system. A minimum configuration (unit processor) of the same system would consist of a processor and two memory modules (65K).

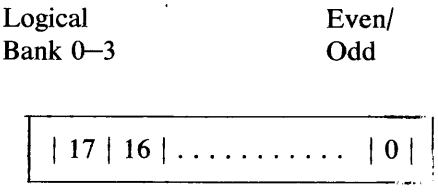
The 1108 Executive system views the minimum configuration as a subset of the more general multisystem configuration. To better understand the multiprocessor system, it is helpful to think of it as composed of many distinct modules or system components which are functionally, logically, and electrically independent. Any system component can be removed from the system, resulting only in a degradation of system performance, and not total system destruction. The removal of the system component is accomplished by a combination of software and hardware actions. (This will be discussed further under System Availability.)

The following system components can be interconnected into a variety of pre-specified system configurations.

1108 II processor

This processor is functionally identical to the 1108 Processor except that an additional mode of main memory addressing was introduced, and another instruction was added to the Unprivileged Set.

Additional mode of main memory addressing.— To the programmed overlapped addressing of the 1108 Processor was added the interleaved capability with an extension of this addressing form to include 262K, 36-bit main memory words. This interleaving is accomplished after the basing and indexing operation by decoding the upper two bits and the lower bit of the absolute memory address.



Address Field

The upper two bits specify the logical bank and the least significant bit of the address field specifies the even or odd module within the logical bank. Each logical bank contains 65K of 36-bit words. The interleaving feature permits optimum simultaneous execution of a common area of procedures by two

(or more) processors. The degree of interleave is a trade off between simultaneous access capability and hardware functional capability during times of main memory hardware malfunction.

Addition of the test and set instruction to the unprivileged set.—The function of this instruction is to permit queued access to common procedure areas that cannot be executed simultaneously by two or more processors. This function is accomplished by testing the specified main memory cell and setting it during the same extended memory cycle.

Memory Cell $2^{30} = 1$	
Yes	No
Interrupt to Loc. 164	Take NI

Consequently, this common area of procedures is protected from simultaneous entrance by virtue of the fact that all references to it must test the cell associated with that area. If simultaneous access is attempted, an internal interrupt to the Executive System is effected. These restricted entrances into procedure areas will occur most frequently in the Executive area during assignment of, or changes in, priority of system tasks. It is beyond the scope of this paper to delve into discussions on re-entrant or pure procedure, and conversational mode compilers. They are, however, planned as standard software packages for 1108 Multiprocessor Systems.

Main memory

Main memory is provided in 65K word increments, expandable from a 65K minimum to a 262K maximum configuration. The main memory cycle time is 750 nanoseconds. Eight distinct paths are provided for each Processor and IOC, thus enabling each of these components to address the maximum memory configuration. Main Memory is composed of eight separate 32K modules, thus eight simultaneous references could occur in a system configuration containing the maximum memory, processor, and IOC components.

Multiple module access unit (MMA)

This system component provides multiple access to individual memory modules. A pair of MMA's provides access to one logical bank of 65K words. A maximum of five paths to each module exists. Priority resolution between simultaneous memory requests is accomplished in this unit. The Processor-IOC-Memory interface is request-acknowledge logic. Therefore, at a time when queues develop at a module interface a Processor or IOC can be kept waiting while memory requests are serviced on a predetermined priority basis. This priority is so arranged that IOC's are serviced first due to the inability of most peripheral devices to "wait" for any extended period. Requests are serviced in the following manner. Priority is sequenced from left to right in each class with Class 1 having preemptive priority over Class 2.

$\underline{IOC_0\ IOC_1}$	$\underline{P_0P_1P_2}$
Class 1	Class 2

Input output controller

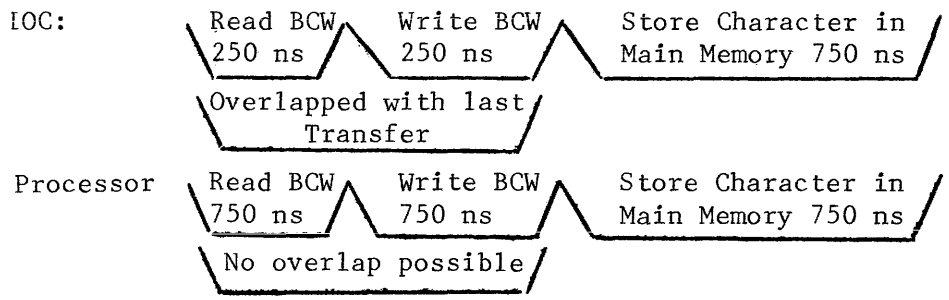
The main function of this unit in the system is to provide the capability for simultaneous compute and input/out data transfer operations. It also enhances the real-time capabilities of the system by providing high speed Externally Specified Index operations. Other primary functions include system channel expansion capability and data buffer chaining. The IOC is functionally similar to the input/output section of the 1108 II Processor. It does, however, exhibit the following advantages over the processor as a memory to peripheral subsystem data transfer device.

(1) Requires *no* interruption of processor computational capability to execute data transfers.
(2) Possesses data buffer chaining capability.
(3) Performs ESI data transfers to main memory utilizing only one main memory reference versus three main memory references for a processor executed ESI transfer.

The operations of the IOC are initiated via control paths from each of the three processors in the system. A channel from each processor is required as a control path for the issued commands and the resulting termination interrupts. The commands specify where function and data buffer control words are located in main memory; it is then up to the IOC to procure and issue functions as well as initiate, control, and monitor the resulting data transfers. Upon completion of an operation, a pre-specified processor is interrupted and notified that the channel has been terminated. The status associated with this termination interrupt is automatically stored at a pre-specified location in main memory by the IOC.

The control memory of the IOC is 256 words (expandable to 512 words). Its cycle time is 250 nanoseconds.

This high speed internal memory enables the IOC to hold up to 448 ESI buffer control words internally. During an ESI transfer the IOC need only reference this memory for its control words and therefore requires only one main memory reference at the time that data is to be transferred. In contrast the processor must reference main memory three times for the ESI buffer control word (as shown below):



The maximum aggregate data transfer thruput of an IOC is 1.33 million words per second.

Multiple processor adapter

This functional unit provides multiple access capability to a peripheral subsystem. It honors requests for access on a first come-first served basis. The unit has two basic modes of operation, one for Internally Specified Index (ISI) operation and the other for Externally Specified Index (ESI) operation. Up to four input/output channels from any combination of Processors and IOC's can be connected via the MPA to a peripheral subsystem.

In the ISI mode the MPA will "lock" onto the first channel that issues a function and will continue servicing that channel until one of two contingencies occur. They are:

(1) An External Interrupt is issued by the Peripheral Subsystem, or
(2) A Release Control function is issued by a Processor or IOC.

In the ESI mode the MPA will remain locked onto a Communications Subsystem until another channel issues an External Function. In the normal ESI mode the MPA remains continuously dedicated to the initiating input/output channel. In the case of a channel malfunction, control of the Communications Subsystem can be switched to another channel connected to that MPA. The switching is accomplished by a function being issued by the channel wishing to take control.

Dual access devices

In addition to providing multiple access to individual subsystem control units, the capability also exists for dual access to certain devices by two control units. The devices provided this capability are Tapes, FH Drums, and FASTRAND Drums. These devices have the capability for resolving simultaneous requests for service from two independent control units. If one of these devices is busy, a busy status is presented to the other control unit. Upon completion of the current service request, the "waiting" control unit is granted access. Further discussion of simultaneous service requests to a device is included under the System Control section of this paper.

System-control (software)

It is obvious from the preceding hardware discussion that many hardware configurations exist and also that many alternate paths between system components are available. However, software and system performance considerations makes it advantageous to pre-specify legal system configurations, and the primary data flow paths as shown in Figure 2.

System control is maintained by the Executive (EXEC) software system. The EXEC can be executed by any processor in the system. Each, in turn, acting as the EXEC processor will inspect the list of current activities and select a task to be done. As previously stated in the hardware discussion, the EXEC processor may interlock an area of critical common procedures or data during task assignment or facilities assignment operations. Input/output operations are normally initiated using the primary data flow path shown in Figure 2. To increase system performance the EXEC will also assign another path to a string of dual access devices such as the FH Drums. This secondary path may also serve as the primary path in the case of malfunction. Secondary paths to single access device subsystems (such as a card subsystem) are also assigned during system generation so that they may function as primary paths during IOC or Processor malfunction. The EXEC routines interfacing with the resident worker programs are reentrant in design. For minor tasks requested of the EXEC, many of the routines are totally re-entrant.

Others, when in the multiprocessing environment, will queue the worker program requests where serial processing in a particular area of the EXEC is required. At the lowest level, the EXEC must queue interrupts and issue input/output service requests on the pre-designated processor. Above this

basic level, any of the available processors can perform a given task with selection based on the priority of the task currently being executed by the processors.

System performance

Several methods have been used in an attempt to quantitatively express the additional increase in computational performance afforded by the addition of one or more processors to the basic system.

Most of these methods consist of complex mathematical expressions and simulation models or a combination of the two in order to depict all of the factors of the configuration and their complex inter-relationships.

The simple expression presented below is only intended to show the primary effect of the factors.

$$N = \frac{P \times 10^6}{C+Q+D+E} \text{ instructions per second} \quad (1)$$

Where,

P = number of processors
C = cycle time of memory (the memory itself)
Q = delay due to queues at memories
D = delays due to hardware (MMA, etc.)
E = time added due to extended sequence instructions and C, Q, D, and E are in microseconds

C and D are straightforward factors; E is best gotten from the Gibson Mix or the like, and Q is the most difficult number to arrive at.

As an extreme case, consider the 1-processor configuration with a 750 nanosecond memory, no queues at memory and no hardware delays. The maximum rate of instruction execution with no extended sequence instructions would be,

$$N = \frac{1 \times 10^6}{.75} = 1.33 \times 10^6 \text{ instructions per second}$$

(theoretical maximum-one processor)

The maximum rate of instruction execution with extended sequence instructions would be,

$$N = \frac{1 \times 10^6}{.75+.300} = 0.95 \times 10^6 \text{ instructions per second}$$

(for one processor)

As an example of a 2-processor case; set C=.750 and D=.125. An estimate for E would be about .300.

Through the medium of various simulations of the 2-processor case, a value of Q=.050 may be arrived at.

Hence,

$$N = \frac{2 \times 10^6}{.75+.050+.125+.300} = 1.63 \times 10^6 \text{ instructions}$$

per second (for 2 processor-multiprocessor system)

Therefore,

$$\text{Gain for 2nd processor is } \frac{1.63-0.95}{0.95} = 0.71$$

The value of E may be adjusted through a small range to reflect the type of problem mix. Q is influenced most strongly by the configuration, the I/O load, and the intelligence of the assignment portion of the Executive. Q is also somewhat interdependent with E.

System availability

Central System availability is assumed to be required on a continuing basis. To achieve this goal modular independency is stressed throughout the system. Each system component is electrically and logically independent. The same provision is made regarding cooling and power supplies. Provision is also made for disconnecting any malfunctioning components without disrupting system operation.

This disconnection is initiated either by the Executive System or the operator. The component in either case is first "program disconnected" by the Executive System and then the operator is informed via the system console. System operation will then degrade to a level which is dependent upon the malfunctioning units normal contribution to system performance.

The Availability Control Unit (ACU) and the Availability Control Panel (ACP) shown in Figure 3 provide the capability for isolating components from the remainder of the operational system for purposes of test and maintenance. Within this framework two other important features are included. They are: Partitioned System capability, and an Automatic Recovery System (ARS).

The ARS is basically a system timer; it is dependent upon the Executive System to reset it periodically (millisecond to several second intervals). If it is not reset, a catastrophic malfunction of the Executive System is assumed, and an initial load operation will be initiated by the hardware using a prespecified portion of the total system. An example of a pre-specified split in the system configuration is shown in Figure 4.

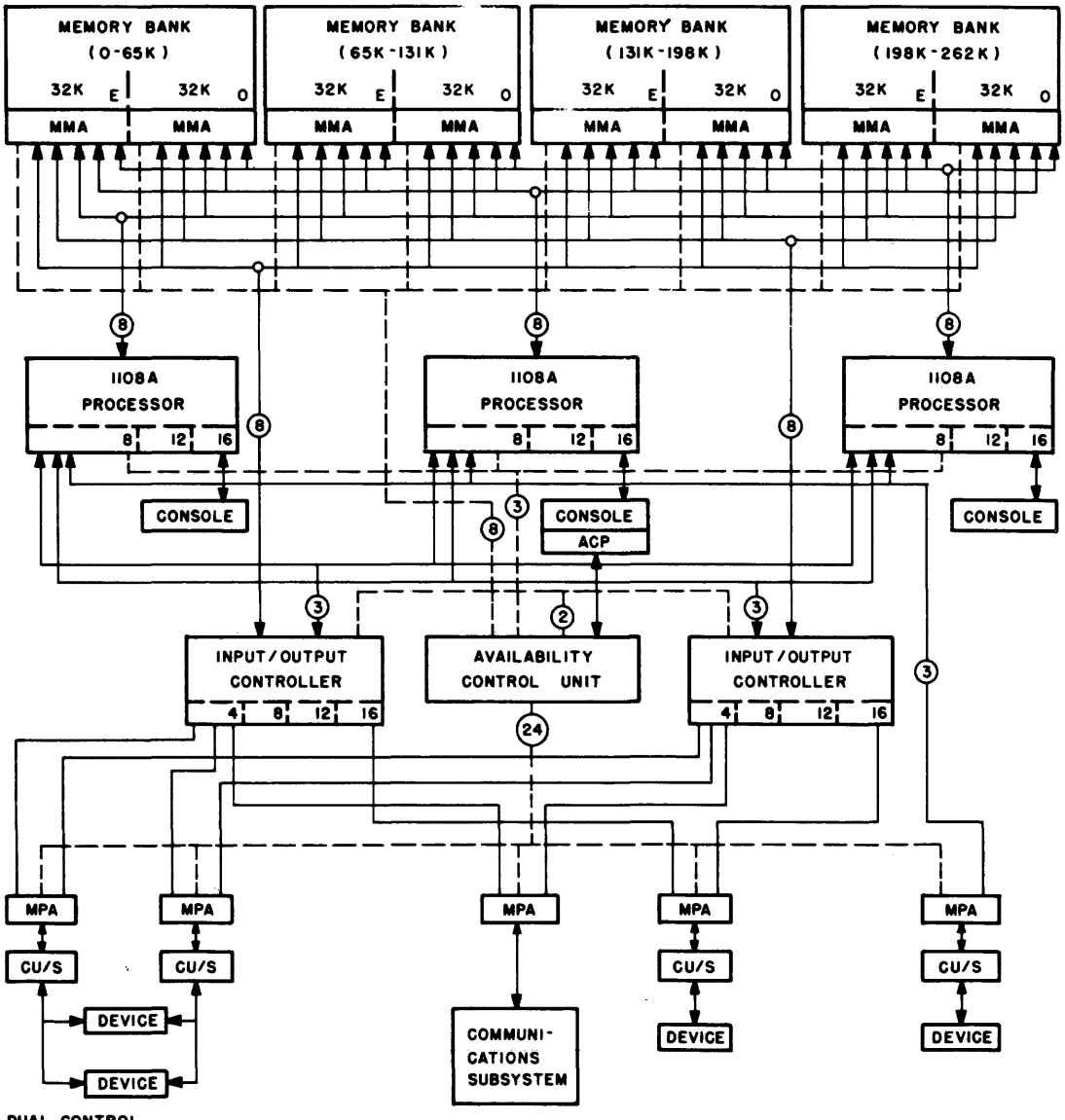
For the system shown in Figure 4 the left-hand portion would try to initial-load the Executive Sys-

tem first. If the malfunction still exists (the system timer would trigger again) then the right hand portion of the configuration would attempt the initial load operation. Assuming one malfunction, one of the two configurations will be operational.

This then is the Partitioned System recovery concept implementation. The system can also be partitioned manually by selection. The partitioning is pre-specified, and the initiation only serves to activate the pre-selected partitions.

SUMMARY

The 1108 Multiprocessor System is a large scale central data processing system which functions equally as well in a real-time, demand processing application as in a scientific batch processing environment. The symmetrical system configuration permits a high degree of parallel processing of tasks by all system processors. Modularity of system components provides for total system availability as well as ease in expansion capability for the future.



RECOMMENDED CONFIGURATIONS

PROCESSORS	1	2	3	1	2	3
IOC'S	1	1	1	2	2	2
MEM(MIN)	131K	196K	262K	131K	262K	262K

Figure 1 – Univac 1108 multiprocessor system – maximum configuration

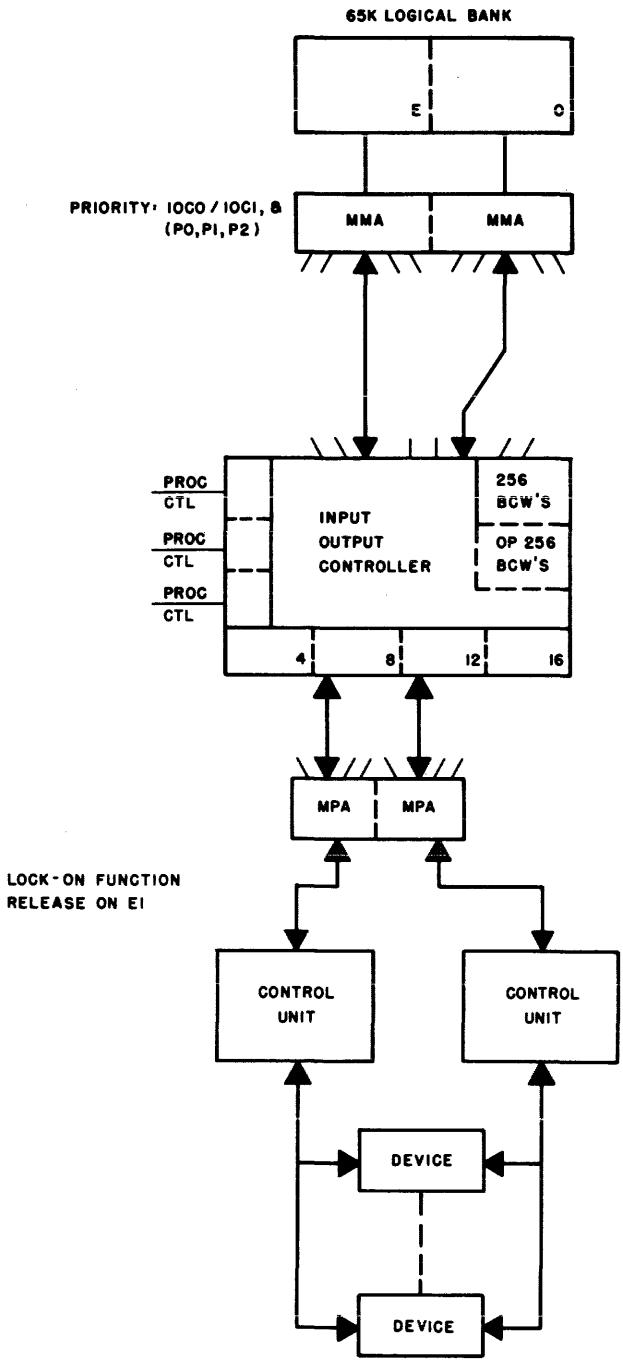


Figure 2—Primary data flow

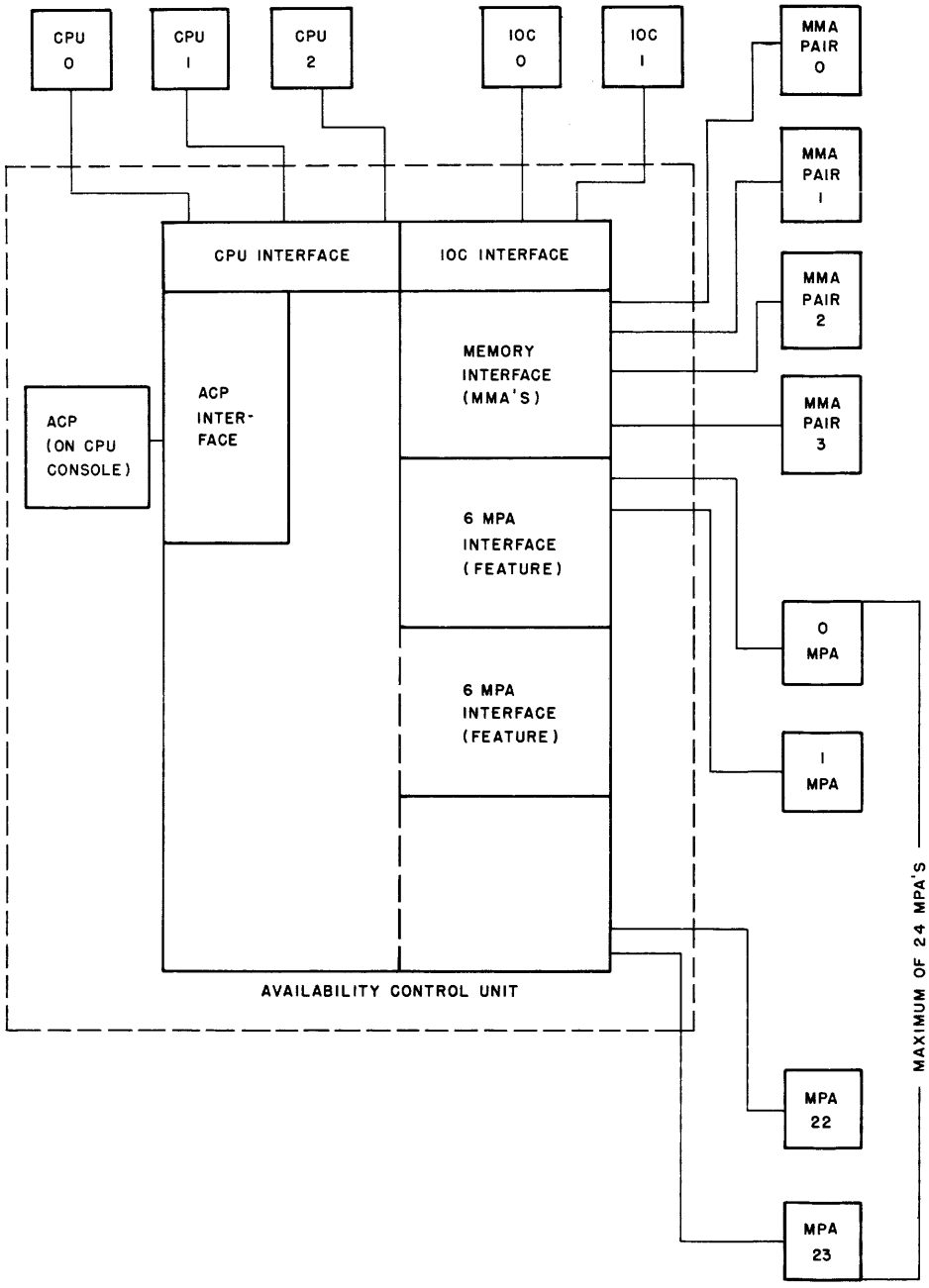


Figure 3 – Availability control unit

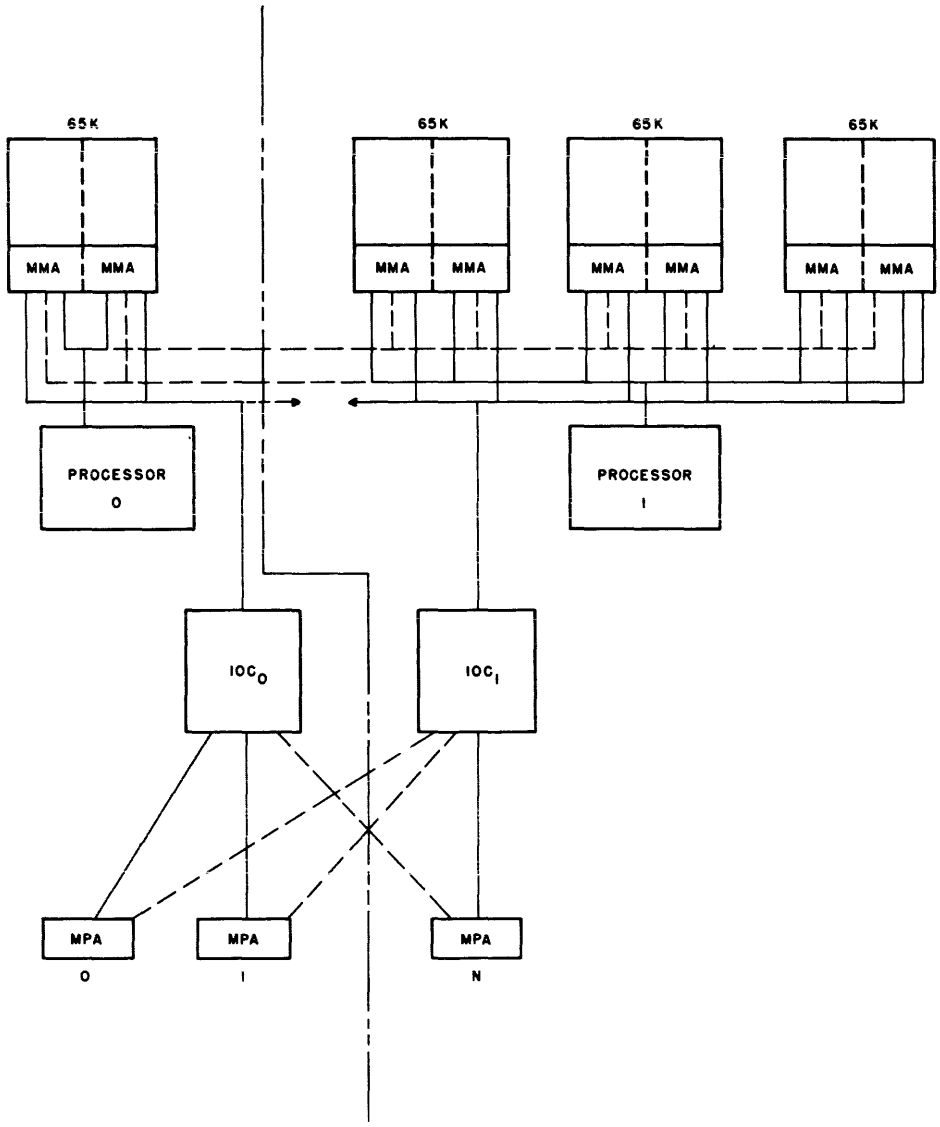


Figure 4 – Partitioned system

Considerations in block-oriented systems design

by D. H. GIBSON
Systems Development Division
International Business Machines Corporation
Poughkeepsie, New York

SUMMARY

The feasibility of transmitting blocks of words between memory and CPU is the subject of this study. The question is pertinent to the design of very fast computing systems where the nanoseconds to traverse a few feet become significant. There is intuitive advantage to transmitting blocks of words, rather than a word at a time. The initial access time due to physical distance, effective address mapping, and priority is a few hundred nanoseconds. If this time could be prorated against several words, then the *effective* access time could be reduced to a few tens of nanoseconds. The question is, of course, can the extra words be useful to the CPU?

This question was explored in a simulation model driven by customer-based IBM 7000 series data. The simulation results indicate that blocks of 4, 8, or 16 words, transmitted to a local storage of 2K to 4K words, will adequately prorate memory access time. With this configuration, block transfer is seen to be an efficient memory access method which can provide high performance, superior to single-word access.

Study goals

The classical CPU reference to memory is for one word, usually for one instruction or one piece of data. Block transfer provides more words than a CPU asks for. A CPU designed for block transfer will save the extra words locally and if it should refer to one of these words, it will not be necessary to go back to memory. On the other hand, if the next CPU reference is for a word in any other block, and the CPU can only save one block, then it will be necessary to go back to memory and waste the transmission time needed to bring over the extra words.

The purpose of this study is to investigate the usefulness of these extra words to the CPU. Specific questions to be asked are:

(1) How many extra words in a block are useful?
(2) How many blocks should the CPUsave locally?
(3) When the CPU must replace one of the blocks it has saved locally, how should that block be chosen?
(4) How does the type of program running in the CPU affect the usefulness of block transfer?

The block transfer system is shown in Figure 1.

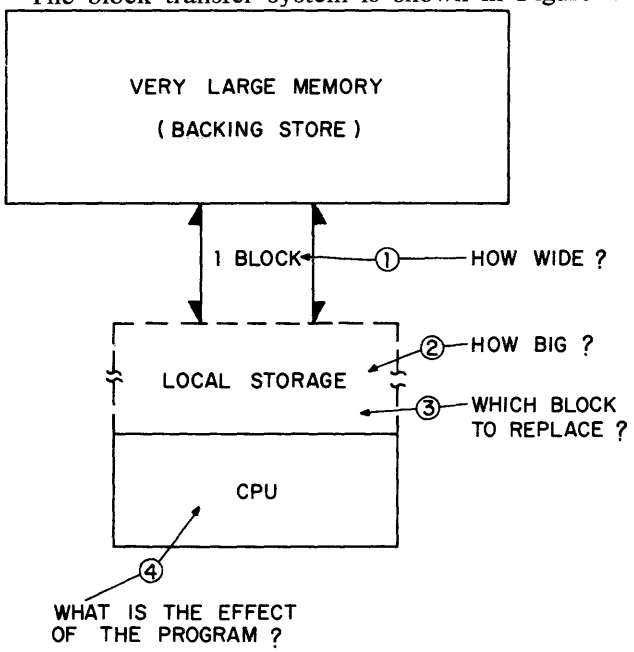


Figure 1 – Block transfer system

Methodology

The pattern of CPU references to memory will determine the usefulness to the CPU of extra words in a block. Conceptually, this pattern could range from random to sequential. In this study, we will first consider the ends of the range – the pure patterns of randomness and of sequential references. Pencil-and-paper calculations suffice to answer the questions posed. The study will then use customer-based IBM 7000 series data to examine addressing patterns on the scale between random and sequential. A computer program aids in the analysis of these other patterns.

Analysis of a particular addressing pattern consists of simulating a block size, a local storage size, and a particular algorithm for choosing the block to be replaced when the local store is full (See Figure 2).

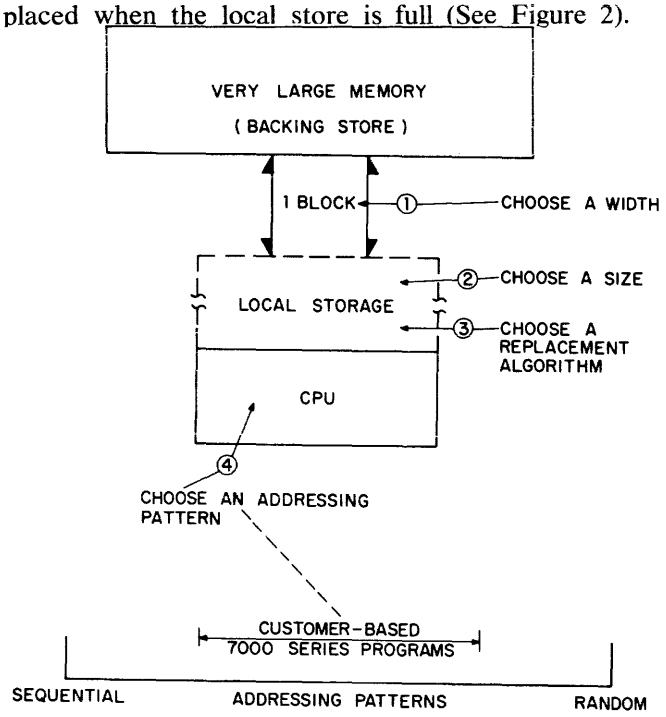


Figure 2 – Analysis of address pattern

The basic number resulting from an analysis is the number of references not found in the local storage; i.e., the number of times a block is transferred to the local store.

Scope of the study

Approximately 600 analyses were run during the course of the study. The block sizes were varied in steps of powers of two, from 4 words per block to 4096 words per block, the local storage sizes were varied from 32 words to 8192 words, and the replacement algorithm was varied over about 15 basic algorithms with several subvariations.

The available addressing patterns were derived from twenty 7000 series customer programs obtained from the field. Each program ran approximately three million address references, which were sliced into 200,000 reference sequences. Thus some 300 address patterns were available for study (See Figure 3).

Address patterns

A random addressing pattern is a sequence of CPU references to memory in which any address is equally likely to occur at any point in the sequence. For such a pattern, the probability of not finding the desired word in local storage exactly equals one minus the ratio of "size of local store" to "size of backing store." Since any word is equally likely to be referenced by the CPU, variations of block size and replacement algorithm have no effect on the number of word references not found in the local store.

A sequential addressing pattern is a sequence of CPU references to memory in which any address is exactly one word away from the preceding address at any point in the sequence. For such a pattern, the probability of not finding the desired word in local storage is exactly the inverse of the block size. Variations of local store size and replacement algorithms have no effect on the number of interest here.

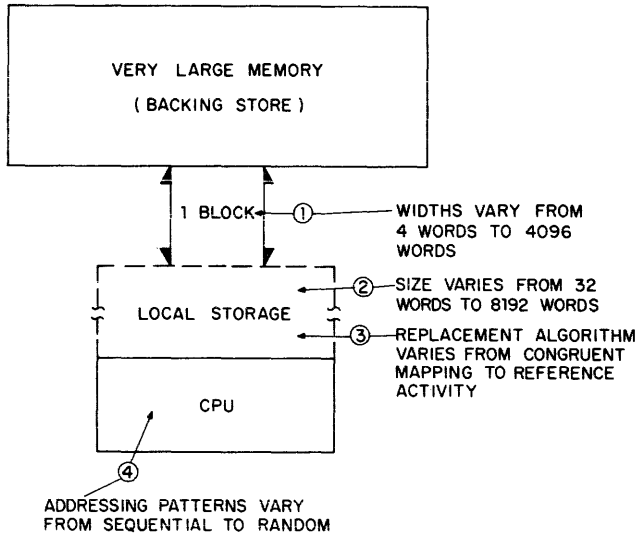


Figure 3 – Scope of study

A set of customer-based 7000 series addressing patterns is neither random nor sequential. The range of results shows two distinct types of patterns. The probability of not finding the desired word in local storage is most often about 0.015, although for a given local store size and block size there are a significant number of addressing patterns for which the probability is approximately 0.075. Variations of block size, local store size, and replacement algorithm affect this probability, as illustrated by the following discussion.

From Figure 4, 76% of the addressing patterns examined have a probability of 0.0275 or less that the desired word is not in a local store of 2048 words, for a block size of 16 words.

Extra words in a block

The usefulness to the CPU of extra words in a block decreases as the block size increases. All other things being held constant, the smaller the block the better. Obviously, however, all other things are not equal; since, for a given technology, there exist a

fixed time per access and a fixed time per word transferred.

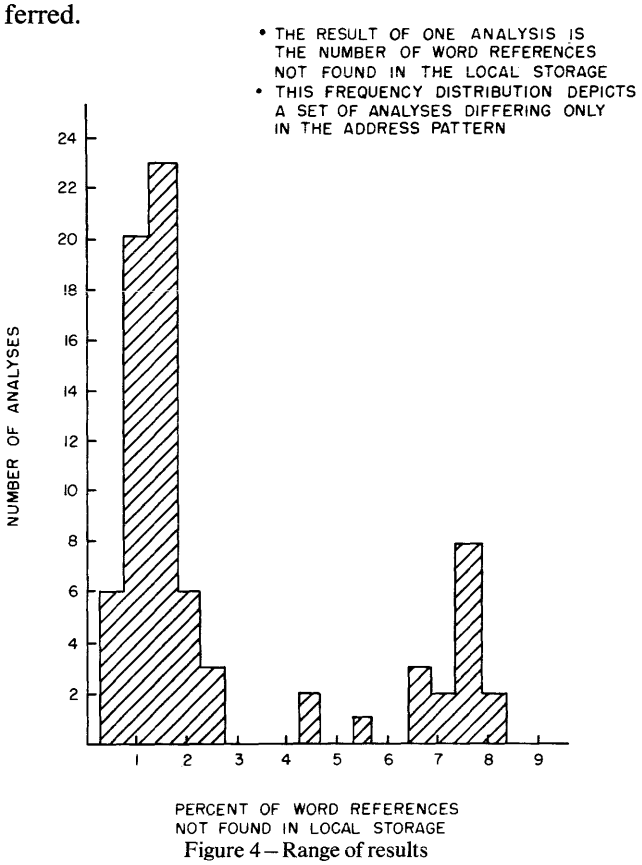


Figure 4 – Range of results

As may be seen in Figure 5, the customer-based 7000 series data indicates that doubling the block size typically doubles the number of words transferred. On the other hand, for a given replacement algorithm and a given local store size, changing the block size does not significantly change the number of accesses outside the local store. Indeed, in the case shown, where the local store is significantly less than what is required to contain the total program, a large block size will produce more traffic between the local store and the backing store than between the CPU and the local store.

No data are available for block sizes smaller than four words. It would be expected that the "number of accesses outside local store" would increase for the smaller block size, while the "number of words transferred to local store" would continue to decrease. In the special case of a one word block, these numbers would be equal to one another, and would equal the number of unique words required by the CPU (about 6000 for this addressing pattern).

Blocks to be saved locally

The typical program has a minimum of four centers of activity; viz., the instruction area, two operand areas for source data, and one operand area for sink data. Theoretically, then, it is desirable to save at least four blocks locally to the CPU.

The customer-based IBM 7000 series data indicates that many more than four blocks should be saved for efficient operation. The different addressing patterns seem to have a commonness in that 128 or more blocks of 16 words each should be saved in local store to assure finding most of the referenced words already available.

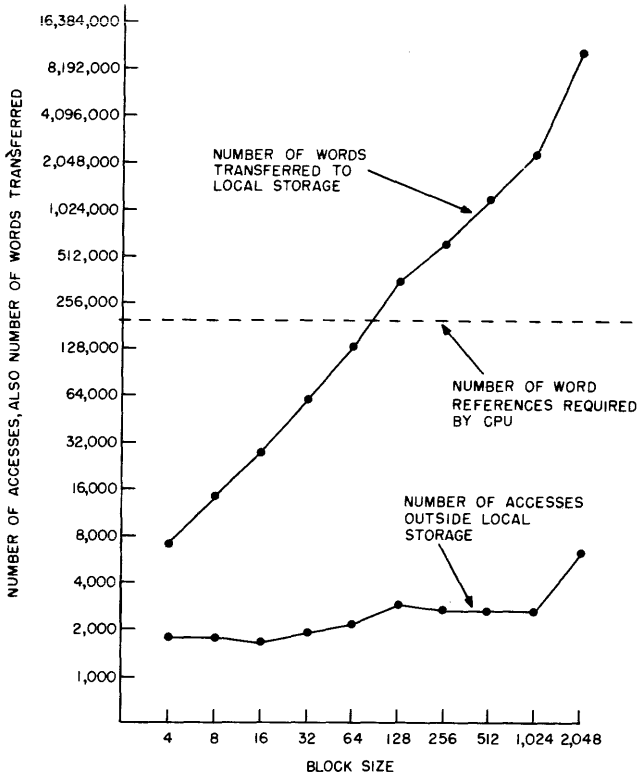


Figure 5 – Block sizes

As the block size increases, the local store size should increase, although the number of blocks that must be saved decreases. The product of block size and number of blocks equals the size of the local store in words. This product is smallest when the block size is smallest. The four addressing patterns shown in Figure 6 are unequal in numbers of blocks referenced, yet converge to a low percentage of outside references for 128 or more blocks. This would indicate that a local storage of 2K words would be suitable for a block transfer system design.

Choice of block to be replaced

The local store contains blocks of information which must be replaced when a desired reference is not found therein. Suppose the processor, upon making reference to the local store for a word, finds that the word is not there. The processor then chooses a space in the local store to hold the block containing the desired word. That space is then examined to determine whether or not it is occupied and appropriate

action is initiated. Finally, the block containing the desired word is moved from backing store to the chosen space locally. The algorithm used to choose the space in local store is called the replacement algorithm.

The data used in this study indicates that the replacement algorithm is of second-order importance to the addressing pattern of the running program in determining the number of references not found in local store. Further, there seems to be no one "best" algorithm for all addressing patterns. Rather, a given algorithm is best for one pattern, and not best for some other pattern.

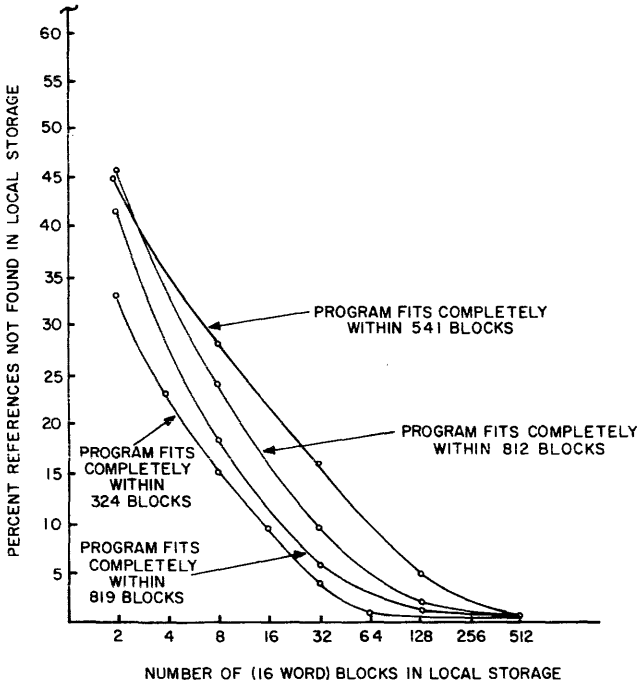


Figure 6 – Local store sizes

The replacement algorithms divide into three classes. The congruent mapping algorithm is probably cheapest to implement but makes least efficient use of local store. A modified version of this algorithm will cost more but can improve efficiency. This class of algorithm uses a binary decode of the address to choose the space in local store. The other two classes use an associative search to choose a space. The "first in, first out" algorithm searches the available space to find the space first filled (i.e., longest ago filled). This class of algorithm is based upon only the blocks in local store and does not deal with block history in the backing store. On the other hand, the third class of algorithm deals with reference activity information stored in extra cells in the backing store; e.g., the total number of times a block was referenced for all stays in local store might be stored in backing store, and used whenever the block is in local store on the replacement algorithm.

Figure 7 indicates that the choice of replacement algorithm for a given system should be dictated at least as much by available technology as by theoretical properties of the contending algorithms.

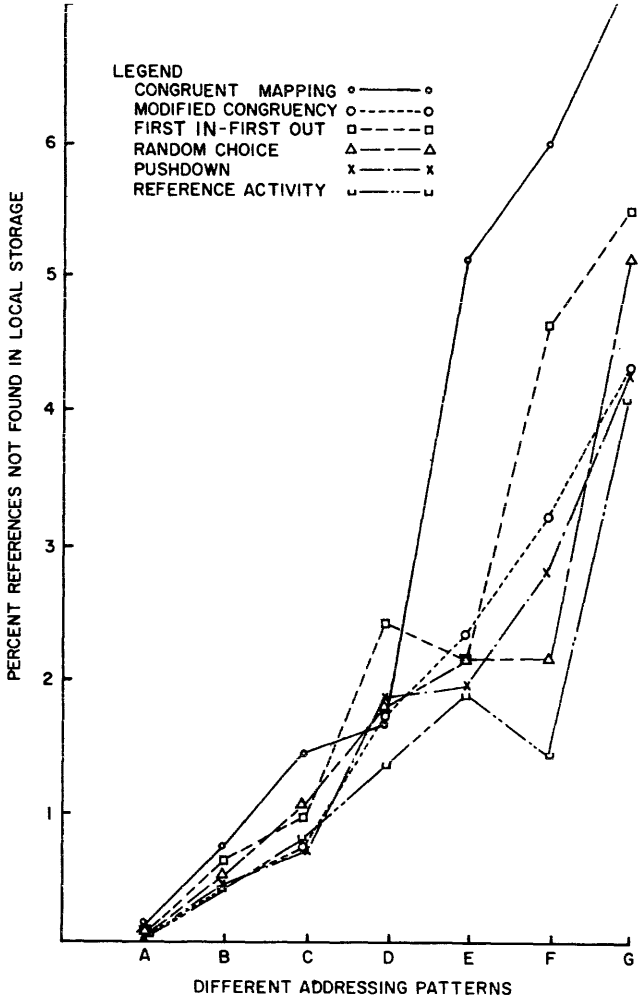


Figure 7 – Replacement algorithm

Program size vs. block transfer effectiveness

The usefulness to the CPU of block transfer is clearly a function of the program running in the CPU, if by "program" we mean "address pattern." No other meaningful description of "program" can be correlated to the swapping activity in a block transfer system. The trace of a single FORTRAN compilation illustrates the inadequacy of program size as a determining factor of swapping activity.

Figure 8 emphasizes the changing address pattern within a single "job." Time has been sliced into units of 200,000 references. The upper chart indicates the changing size of storage used by the compiler, varying from a low of 6.4K to a maximum of 32K in the tenth time slice. This chart is detailed to indicate relative storage required by data and by instructions. The

lower chart indicates the percentage of references not found in local store during the various slices. Note particularly that the greatest incidence of going outside the local store occurs when the used storage size is smallest (during the fifth time slice, when unique words used is nearly minimum).

From this and many similar programs, it is concluded that the program size is of second-order importance to the addressing pattern in determining the effectivness of block transfer.

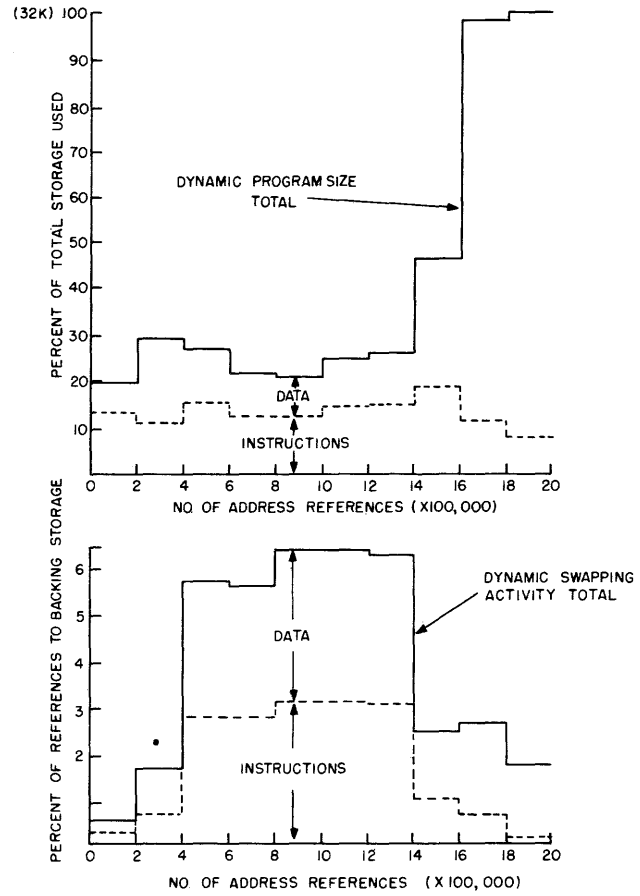
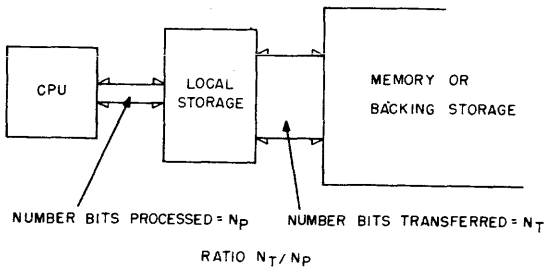


Figure 8 – FORTRAN compiler compiling 600-card program.

Traffic in a block transfer system

One intuitively feels that the number of bits transferred per second is highest between the CPU and the local store, with a significant lower bit rate between local store and backing store, and a lower rate still between backing store and auxiliary storage. This type of bit traffic is certainly desirable when one considers the bit rates of the various devices which constitute the typical storage hierarchy.

The 7000 series data shows that to achieve this desirable traffic rate, block size and local store size must be carefully chosen. While Figure 9 illustrates a local store/backing store hierarchy, the numbers have equally valid interpretation for the classical core/drum hierarchy. Thus, if a drum is accessed for 1024 words per block, and if the program has only 4096 words allotted for use in core, then 1.9 words will be transferred from drum to core for each word transferred from core to CPU.



BLOCK SIZE \ LOCAL STORAGE SIZE	32	64	128	256	512	1024	2048	4096	8192
16	6.5	1.9	1.4	1.0	.70	.17	.10	.045	.033
32		12.1	3.0	2.0	1.2	.37	.16	.072	.039
64			23.2	4.5	2.5	1.2	.27	.14	.043
128				44.6	7.0	3.1	.45	.22	.073
256					84.1	11.3	2.5	.35	.09
512						157.	6.9	.73	20
1024							24.6	1.9	.42
2048								7.9	1.2
4096									3.7

Figure 9 – Transfer channels.

Traffic density is most often favorable for small block sizes. If a large block must be used, then a corresponding very large local store must be chosen.

Block transfer system performance

The "number of references not found in local store" can be translated to approximate system performance for given hardware implementations. This is most readily done by assuming that the CPU generates a word request for every potential local store cycle. The limit to the system performance is then set by the local store cycle time, and would be reached if the "number of references not found in local store" were zero.

As an example, suppose we compute the system performance for "number of references not found in local store" equal 275 out of 10,000. We will assume a local store cycle of 50 nanoseconds, and we will assume that the distance $D_A = D_C$ (as shown in Figure 10) corresponds to 150 nanoseconds. (Note that this figure includes time for priority determination at the backing store, and for dynamic relocation.) We will assume a backing store cycle of 800 nanoseconds with a block readout of 16 words. We compute as follows:

Time for one block reference
= 150 + 800 = 950 nanoseconds
Time for 275 block references
= 275 × 950 = 261,250 nanoseconds
Time for 10,000 word references
= 50 × 10,000 = 500,000 nanoseconds

Total time for references = 761,250 nanoseconds
Average time per reference
= 761,250 ÷ 10,000 = 76.125 nanoseconds

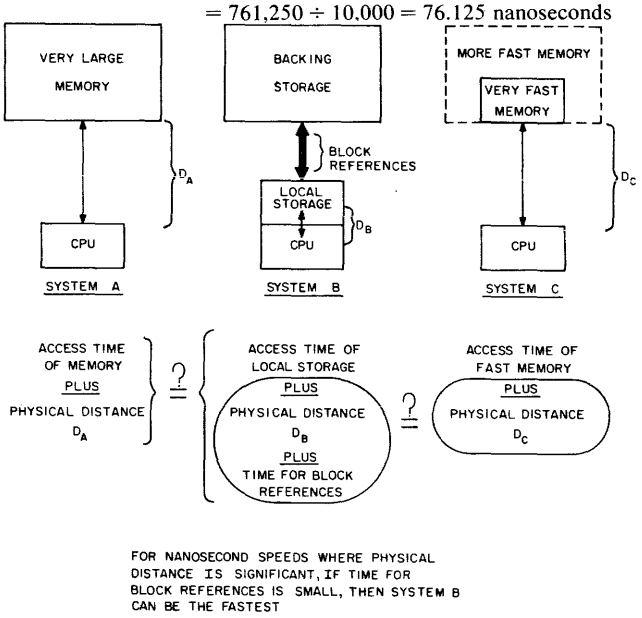


Figure 10—Block transfer system performance.

This is for System B as shown. For System A, the average time per reference is 150 + 800 = 950 nanoseconds. For System C the average time per reference is 150 + 50 = 200 nanoseconds.

The above calculation ignores many important parameters, and may not be used to precisely compare system options. Such comparison is, in fact, the subject of an entirely new study. The author has participated in preliminary comparative studies which show that systems of Type B and Type C are more closely equal in performance than shown here. Nevertheless, the calculation indicates the potential performance advantage offered by block transfer when the percentage of references not found in local store is quite small.

CONCLUSION

For local store/backing store

(A) Small blocks preferred, with 4 word, 8 word, and 16 word blocks proving quite suitable.
(B) Size of local store suitable at 2K to 4K words.
(C) Replacement algorithms can sometimes improve speed.
(D) The swapping activity cannot be correlated to the size of the program.
(E) The backing store/local store/transfer channel traffic and bit rates must relate properly.
(F) The address patterns of the various programs produce the most significant variation in swapping activity.

REFERENCES

1 L A BELADY
A study of replacement algorithms for a virtual storage computer
IBM system journal
Vol 5 No 2 1966

2 T KILBURN
Data transfer control device
U S Patent No 3,218,611 November 16 1965

3 T KILBURN
Electronic digital computing machines
U S Patent No 3,248,702 April 26 1965

4 F F LEE
Look aside memory implementation
Project MAC-M-99 MIT Cambridge Mass August 19 1963

5 G G SCARROTT
The efficient use of multilevel storage
Proc IFIPS congress
Spartan Books 1965

6 M V WILKES
Slave memories and dynamic storage allocation
Project MAC-M-164 MIT Cambridge Mass June 22 1964

Intrinsic multiprocessing*

by RICHARD A. ASCHENBRENNER
Argonne National Laboratory
Argonne, Illinois

and

MICHAEL J. FLYNN
Northwestern University
Evanston, Illinois

and

GEORGE A. ROBINSON
University of Wisconsin
Madison, Wisconsin

INTRODUCTION

Intrinsic multiprocessing

Multiple access systems for processing data from on-line experiments or remote consoles are normally treated by time-slicing multiprogramming techniques. A substantial share of CPU time is consumed with program swapping, terminal polling and buffering, I/O control, etc. This increases the overhead time loss and/or decreases the number of terminals a system can accommodate.

Multiple CPU's operating in a switching exchange mode or sharing at least a common memory have also been used to increase reliability in situations where malfunctions are catastrophic or at least economically untenable. Increased supervisor complexity as well as a substantial cost increase is one price paid for this means of achieving reliability.

On the assumption that a substantial increase in throughput must lie in an increased parallelism in the operation of a computer, a technique termed *intrinsic multiprocessing* has been an object of study at Argonne for some time.[1]

IMP

The Intrinsic Multiprocessing (IMP) system is depicted in Figure 1. The system can be considered as a network of several general purpose machines (sequence control units - SCU) (Figure 2) . Its functional operations are, however, distributed throughout the system with time sharing of that part of the hardware which would have infrequent use by a single machine.

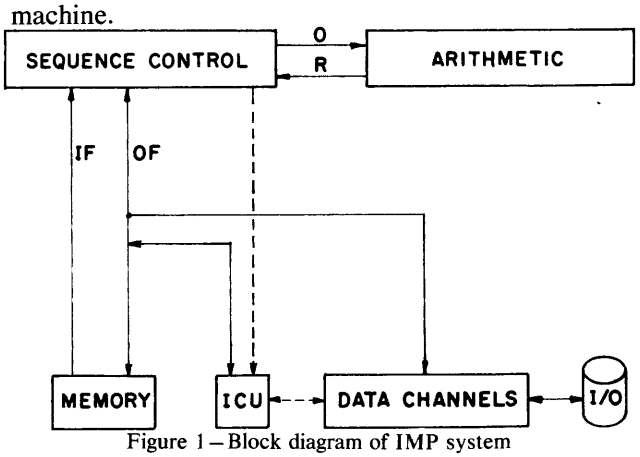


Figure 1 – Block diagram of IMP system

The purpose of IMP is to take advantage of the natural independence of time-shared programs to achieve a maximum in efficient use of the execution hardware and memory of the system. The system consists of n time-phased virtual machines, sharing very high speed execution hardware.

The system's objective is that each virtual machine has performance comparable to conventional large scale scientific computers, for example, the IBM 7090. For purposes of this paper, the IBM 7090 cycle

*Work performed under the auspices of the U. S. Atomic Energy Commission.

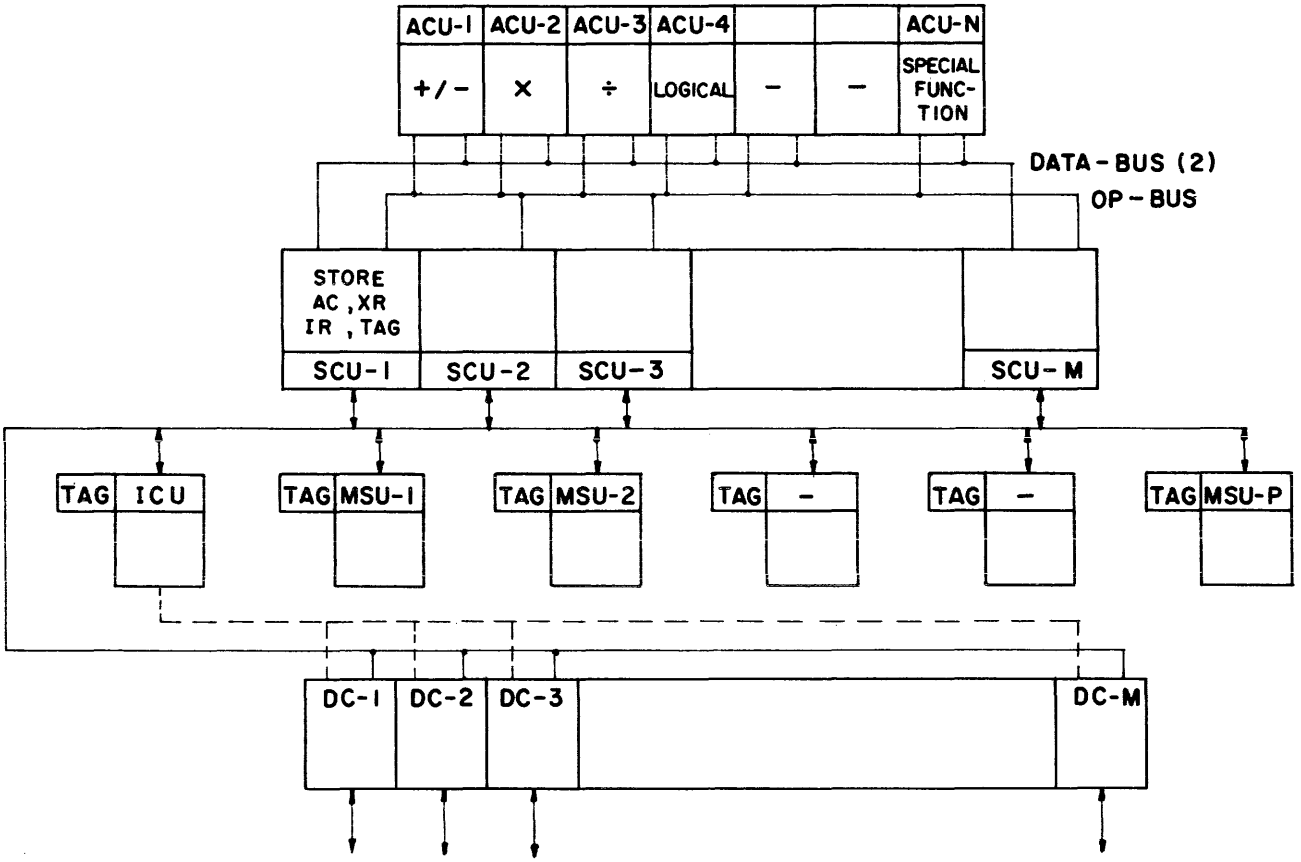


Figure 2—Parallelism in the IMP system

time and instruction set will be used as a comparative classical organization.

Sequence control unit

The sequence control unit serves as the central control for sequencing the execution of a particular job or path of a segmented problem presently active in IMP. Each module is similar in structure but operates simultaneously with, and independently of, all other SCU's.

Basically each SCU consists of a series of registers (see Figure 3). These registers include a store/fetch register, an accumulator, an instruction counter, an instruction register, a memory address register (MAR), and several index registers (XR). The SCU is responsible for the decoding and execution of the load, branch, and store instructions. If a fetched instruction (IF - Figure 1) is not of one of these classes, an operand is obtained (OF - Figure 1) and the instruction must be placed on an operator bus. The operator bus (OP BUS) considered here operates at a 25-ns minor cycle rate. The high order bits of the operation code specify the execution unit which will perform the particular instruction. If that particular unit is free, an accept signal is sent back to the requesting SCU. If the unit is busy, a reject signal is sent back and the SCU waits one major cycle to reinitiate the request. Figure 4 shows the structure of the opcode field of the instruction. All instructions are presumed to be of the one-address class. The SCU must be able to perform tests on the accumulator and the store/fetch register for the execution of the branch instructions. It also has the data paths to carry out the load and store instructions. SCU instructions are distinguished by a "one" in the highest order position of the instruction. If the SCU bit is zero, it is disregarded and the remaining n-1 bits are placed on the operator bus at the proper time. The remaining higher order bits specify the execution unit type. The lower order bits complete the specification of the exact instruction to be performed.

Figure 5 shows the SCU timing and the steps in the execution of an SCU instruction in a highly structured IMP system. For sake of discussion here, we have selected a 2 μs SCU memory. The SCU is operated in a strictly serial fashion (much the same as the 7090). One can, in fact, assume that the data word and instruction words are essentially the same as the IBM 7090. The instruction fetch takes 1 μs. Another microsecond is devoted to regenerating memory. Decoding of the instruction is overlapped with memory regeneration. Decoding involves determina-

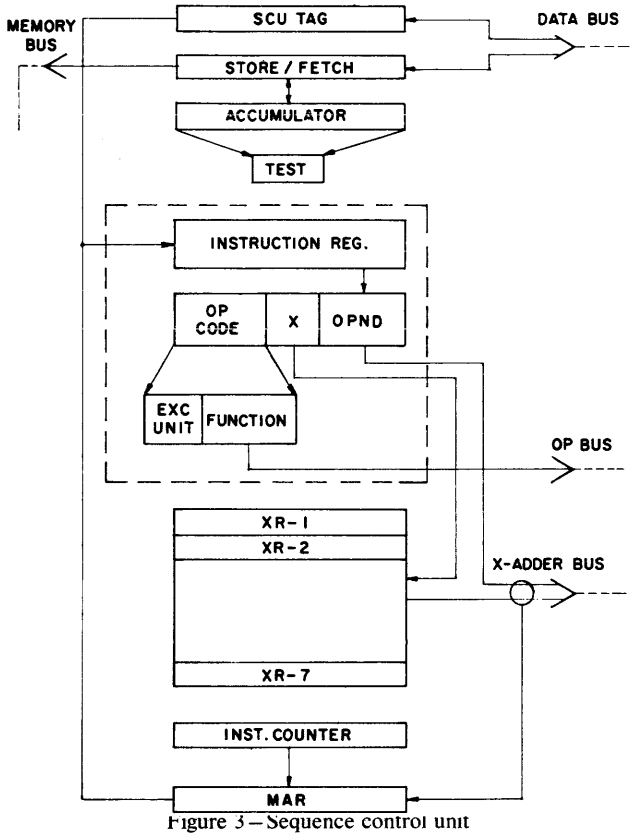


Figure 3—Sequence control unit

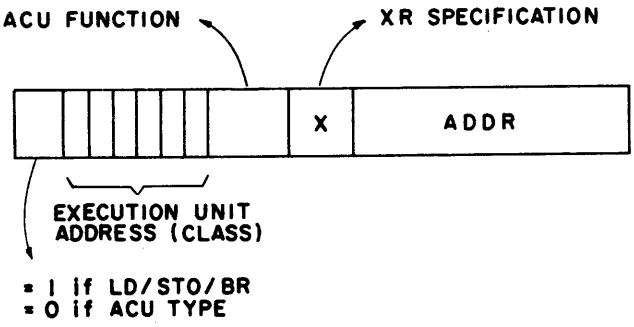


Figure 4—Instruction format

tion of whether or not this is an SCU instruction and, if it is, the exact type. After the instruction is decoded, it is necessary to generate the address of the operand. The operand address portion of the instruction register must be added to the contents of the specified index register. There will be, in general several index adders. Each index adder will perform the additions for either four or eight SCU's.

Upon the generation of the operand address and after the completion of the memory regeneration, an operand fetch may be commenced. Two minor cycles (50 ns) before the end of the operand fetch, service is requested from the unit specified in the higher order bits of the opcode. One minor cycle before the operand fetch is complete, an accept or reject signal is sent to the requesting unit; that is, placed on the bus and sampled by the requesting unit. If the request in honored, the operand is placed immediately on the bus (O—Figure 1). One minor cycle is allowed for entry of the operand into the receiving execution unit. At the same time the operand is regenerated in the SCU memory. Execution is not accomplished in one minor cycle, rather it has a latency of perhaps 200 ns. However, it is able to accept operands at the rate of one pair every two minor cycles. The result is placed back on the bus (R—Figure 1) immediately after the completion of the execution. One minor cycle later the result is latched into the accumulator of the individual SCU. Since there is ample time before the operand regeneration is completed, tests (if any) on the accumulator or storage register may now be performed before the address of the next instruction is generated.

Arithmetic control unit (ACU)

Since the arithmetic control units service all the SCU's, the actual number of each type of ACU module (adders, multipliers, etc.) is determined by three characteristics of the multiprocessor:

(1) The number of SCU's in the system.
(2) The performance degradation of the system due to an excessive queue being formed for any particular type of ACU.
(3) The requirement for special non-standard functional operations.

The second of these characteristics must be considered on the basis of the first as well as the computational bandwidth of the ACU and the instruction mix assumed.

The third characteristic indicates the "open-end" nature of the system. The ACU's are individual units that perform an arithmetic operation or a special functional operation. The ACU's are time-shared among the SCU's, which are, in turn, processing independent programs.

Residue checks in arithmetic ACU's will drop a malfunctioning unit from service until a diagnostic check by the supervisor control and subsequent off-line service is performed. This will tend to minimize downtime while imposing only "graceful" degradation (slow-down) rather than complete shutdown. This philosophy is carried over to other units as well.

The number of units and the specialization is open-ended. That is, new functions can be added as well as duplicate units of the same function.

The only severe requirement placed on the units is that the execution latency or the total execution time be the same for all the units. In practice, this requires that all units match their execution times to

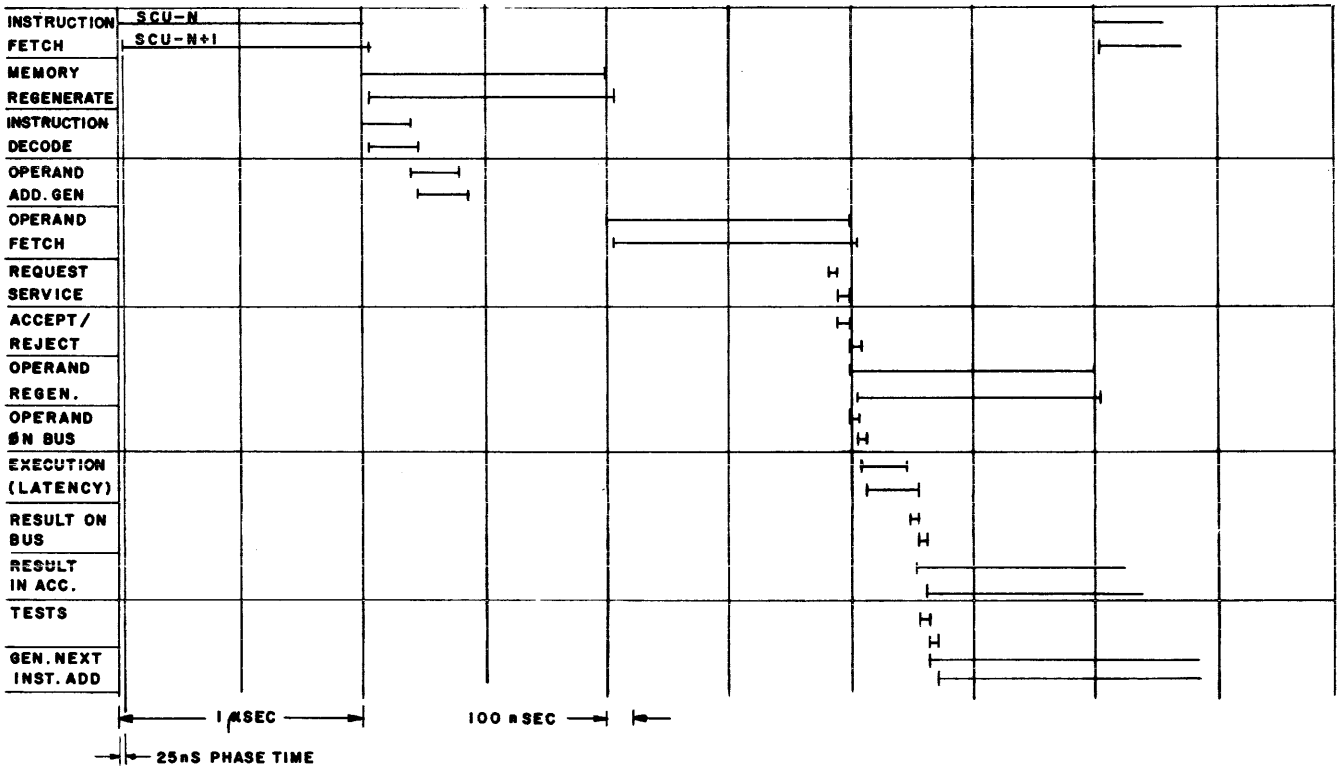


Figure 5—SCU timing chart

the longest unit. This is accomplished by adding delays in short execution unit cases. This requirement can be removed at the cost of increased complexity of control. As a practical restriction, the uniform latency time affects the divide class instruction more than any other. Multiply and add have become increasingly uniform with recent developments of high speed multiplication techniques. On the timing chart (Figure 5), an inherent execution latency of 200 ns is assumed. This is easily fulfilled for a 36-bit data word (as the 7090 would be). For the 64-bit format, full floating adds of 120 ns and full floating multiplies of 180 ns have been reported.[2] Another unusual requirement implied by the IMP execution system is that each of the execution units be "pipelined" to achieve maximum bandwidth.[3] Thus while a floating add takes 200 ns, it goes through a four-stage phasing process. The first phase is instruction preparation, the second phase is preshifting, the third phase is add, the fourth phase is postshifting. A similar arrangement is visualized for the multiply. Logical instructions could be handled at a one minor cycle rate since no part of their execution should demand more than that unit of time. We are presuming that for add and multiply, a time quantum of 50 ns is the minimum to perform any one phase of the execution and hence limits the bandwidth of that unit. The pipeline techniques considered here are similar to those discussed in the literature.[2,3] The major difference is the uniform latency requirement and the more extensive use of pipeline for the IMP system. Depending on the usefulness in user's environment, additional units such as a variable field length, decimal arithmetic, etc., might be added to the execution bus.

The net effect of this system can best be described by a timing chart (Figure 5). Four microseconds is presumed to be the instruction execution time for a complete instruction for each of the SCU's. If sufficient execution bandwidth is available and if no conflicts between requesting units develop, the system will retire an instruction each 25 ns. This would give the IMP system the capability of handling 160 SCU's with one central shared execution area. Since each SCU has performance in excess of a 7090 (since there are no "long" instruction times), barring execution conflicts, the total effect of performance of the system would be somewhat greater than 160 7090's; probably in the order of 200 7090's.

An important item to note here is that one chooses the exact mix of execution units to ensure efficient computational use of the hardware. If, for example, one finds that much of the execution hardware is not being utilized because of I/O requirements and I/O latency delays, one may reduce the available bandwidth in the execution area or increase the number of sharing SCU's. The only constraint inherent in this system is the fact that the bus phase time must be a divisor of the latency time of execution.

Executive control

Thus far we have discussed only the SCU execution interactions and the IMP efficiency advantage in the execution area. There is another level of efficiency at which IMP operates which, in some respects, is even more significant than execution efficiency. This is in the area of memory and the storage hierarchy and executive control.

The main functions of the executive SCU are dynamic reassignment of memory, scheduling of SCU's (to incoming problems or segmented problem paths), and I/O monitor. Queue tables for I/O and SCU selection reside in the memory module containing the supervisor, as do assignment tables for operation of segmented problem paths. In addition, all transfers between bulk memory and core are under the executive SCU control. One of the SCU's (number 1 for purpose of discussion) is designated as the monitor SCU. The choice is arbitrary as its function is not specialized. Its function is principally to determine whether or not each of the SCU's is busy and the memory requirement of each SCU. SCU 1 is responsible for the generation of a job queue; this queue includes the memory requirements associated with a particular job. SCU 1 is then able to assign to a particular SCU not only the job but an appropriate amount of core storage. This may vary from 4K to more than 64K words, since in addition to the minimum requirements (4K) there is a pool of storage which is capable of being switched between units.

Memory storage units (MSO)

In order to achieve this switching with the storage pool, dynamic allocation is necessary.

The memory modules are segmented into 4096-word blocks individually assignable to an SCU problem path. Memory protection is accomplished by each module having an associated tag. The tag includes a bank address register and a reference "lock" register determining which SCU or combination of SCU's is allowed to access the module. The dynamic filling and flushing of memory is under control of the SCU assigned as system monitor.

The executive control SCU assigns to the other SCU's programs to be executed and storage resources to be employed. To the sequence control unit (or set of such units) assigned to a given problem, the Executive assigns a memory reference "key" number which will accompany all requests for access to memory storage units. The corresponding memory reference "lock" number is assigned to each memory storage unit assigned to that problem. Furthermore, each memory storage unit assigned is given a bank address which will be matched with the high order address bits in a storage access request. Pooling and switching is accomplished in a manner very similar to the sharing of the execution units (by a time phase bus scheme). The high-order part of the tag is essentially the SCU label. The low-order part of the tag represents the high-order bits appended for the memory bank. Thus, if SCU 7 has 32,000 words associated with it, there will be on the memory bus eight independent 4K memory units tagged from 7-0 to 7-7.

The information contained in the first 4K of the 32K block is contained in unit 7-0. Similarily, information in the last 4K of the 32K block is contained in memory 7-7. The monitor interacts with the SCU's only to assign jobs, to note job completion, and to supervise I/O operations. Sufficient I/O bandwidth must also be available to supply the processor requirements.

Inter-communication unit and data channels

Communication between the monitor SCU and data channels or other SCU's is accomplished via the inter-communication unit (ICU) (see Figure 1). An SCU wishing to request simultaneous segmented paths (FORK-JOIN)[4] or I/O operation, etc., first stores the request in its core memory and then executes a Call Monitor instruction. This instruction, in effect, transmits the SCU number to the ICU and causes the monitor SCU to be interrupted at the next opportunity. The monitor SCU then obtains the SCU number by a memory fetch. (The ICU is a simulated memory module with lock 255-available only to monitor SCU.)

The monitor SCU can interrupt any other SCU by transmitting (via a store order) the number of the SCU to be interrupted. The ICU then interrupts the appropriate SCU.

Communication between the monitor SCU and any one of up to 160 data channels is handled in a manner analogous to communication between the monitor SCU and other SCU's. Each data channel, in order to have the capability to execute a channel program, has the following registers: instruction counter, word count, data address, instruction code, tag, and data. Adders to increment or decrement the channel instruction counter, word count, and data address are shared in a fashion similar to the sharing of index adders among SCU's.

SUMMARY

This independent organization of arithmetic, sequence control, and memory units give IMP the capacity for *Intrinsic Multiprocessing*, which can now be characterized by the following properties.

(1) Since ACU's are not assigned to SCU's, the ACU needn't stand idle awaiting operand or instruction fetches.

(2) If a particular problem happens to contain a segment in which occurrences of one operation, say multiply, are very frequent, at most only that problem need be delayed by that condition; the remainder of the ACU's and SCU's continue operation.

(3) The high overall performance of the system is accomplished without any need to depart from strictly sequential programming techniques unless convenient to do so. The n sequence control units can work on different problems if desired.

Two further observations about the IMP system:

(1) Where it is desired to provide multiple access facilities, the independent organization of SCU's and ACU's enables one to consider a system with many more SCU's than could be afforded if each required its own complete arithmetic unit.

(2) It is unnecessary to invest in clever hand-coding or sophisticated compiler-optimization techniques in order to achieve reasonable efficient operation of the system.

ACKNOWLEDGMENT

We wish to acknowledge the assistance and support of W. F. Miller of Stanford University and W. Givens of Argonne National Laboratory.

REFERENCES

1 R ASCHENBRENNER G ROBINSON
Intrinsic multiprocessing
AMD Technical Memo No 121 Argonne National Laboratory Internal Report June 1964

2 S F ANDERSON J EARLE R E GOLDSCHMIDT D M POWERS
The model 91 execution unit
IBM Journal of Research and Development to be published

3 L W COTTON
Circuit implementation of high-speed pipeline systems
Proc AFIPS Fall Joint Computer Conference 1965 p 489

4 W COMFORT
Highly parallel machines
Proc of the 1962 Workshop on Computer Organization, Spartan Books Washington D C pp 126-155 1963

ASP: a new concept in language and machine organization*

by D. A. SAVITT, H. H. LOVE, JR., and R. E. TROOP
Hughes Aircraft Company
Fullerton, California

*The work reported here was supported by the Air Force, Rome Air Development Center under contracts AF30(602)-3279 and AF30(602)-3669.

INTRODUCTION

The Association-Storing Processor (ASP) consists of a language designed to simplify the programming of non-arithmetic problems, together with a number of radically new machine organizations designed to implement this language. These machine organizations are capable of high-speed parallel processing, and take advantage of the low cost of memory and logic offered by large scale integration (LSI).

The ASP concept has been developed specifically for applications having one or more of the following characteristics:

(1) The data bases are complex in organization and may vary dynamically in both organization and content.
(2) The associated processes involve complex combinations of simple retrieval operations.
(3) The problem definitions themselves may change, often dramatically, during the life of the system.

A prime example of such an application is information retrieval, particularly as applied to decision making and intelligence operations.

Present solutions to problems in these application areas frequently suffer from excessive programming costs and time, and often from excessive running time as well. In fact, a major problem in the design of many information retrieval systems is that of trying to insure that the programmers will be able to keep pace with subsequent changes in data organization and problem definition. These difficulties can most generally be attributed, directly or indirectly, to the fact that conventional digital computers, being the sophisticated decendants of automatic calculators, are optimized for the performance of serial operations on fixed-sized arrays of data.

An opportunity is now emerging for overcoming these difficulties by making fundamental changes in the organization of the processors themselves. This is a result of the great progress being made in hardware manufacturing technology. The recent advance from discrete transistor circuits to integrated circuits is about to be overshadowed by an even greater jump to LSI circuitry. This new jump will result in 100-gate and then 1000-gate circuit modules which are little larger in size or higher in cost than the present four-gate integrated circuit modules.

To best take advantage of this opportunity, the authors took the approach of beginning with the development of an appropriate language. In doing this they were encouraged by the extraordinary accomplishments in the handling of complex problems which have been made possible by some of the high-level programming languages. The development of machine organizations designed to implement the language was the second step. Two organizations have been completed to the level of a detailed logic specification. This paper describes the general features of the ASP language and one of the machine organizations. (Reference 1 gives a detailed language definition and a description of the other machine organization.)

PART I - LANGUAGE

The ASP language is a machine-independent programming language which is oriented to facilitate the specification of queries and data-base modifications in information retrieval systems. The objective

has been to minimize the amount of procedure which must be specified to retrieve or modify data. The approach to meeting this objective is based upon the use of a data structure in which the associations between data items are expressed explicitly. The name "Association-Storing Processor" was chosen to suggest the significance of storing the data associations explicitly.

The ASP language is sufficiently problem-oriented to be used by non-programmers, and lends itself particularly well to man-machine discourse in on-line retrieval systems. In addition, its characteristics facilitate the writing of compilers to translate from higher-level problem-oriented languages into the ASP language. A close correspondence exists between the ASP language and query languages, making it relatively easy to write efficient compilers for a wide variety of such languages.*

There are two sets of antecedents for the ASP language. One is the family of list processing languages.[2] In contrast to these languages, the ASP language could be characterized as a *set* processing language. In some respects, the ASP instruction is very much like a set-manipulating version of the string-manipulating "rule" used in COMIT[3] and SNOBOL.[4]

The other antecedent is the work being done by linguists in the field of information retrieval.[5,6] Data associations or relations, similar to those used in the ASP language, are being considered, by workers in this field, as a basic building block for representing textual information.[7,8,9]

Data representation

In the ASP language the basic unit of data is the "relation." A relation consists of three components, an ordered pair of items and a link which specifies the type of association which relates the two items. Simple items and links are character strings. (A compound item is introduced later.) A relation may be written in the form of an ordered triple or a directed graph. For example, two items, A and B, which are related by a link, R, can be expressed as

(A, R, B)

or as in Figure la.

An item of data may be associated with any number of other items, with each association being expressed as a distinct relation. However, no item of data can appear by itself in an ASP expression because, in the ASP concept, an item which is associated with no other item would have no "meaning" (i.e., would convey no information). The directed graph format makes it easy to visualize complex structures of relations involving items which appear in more than one relation. (See Figure 1b.) Each distinct item appears only once in the directed graph representation of a set of relations. The physical orientation of a relation has no significance.

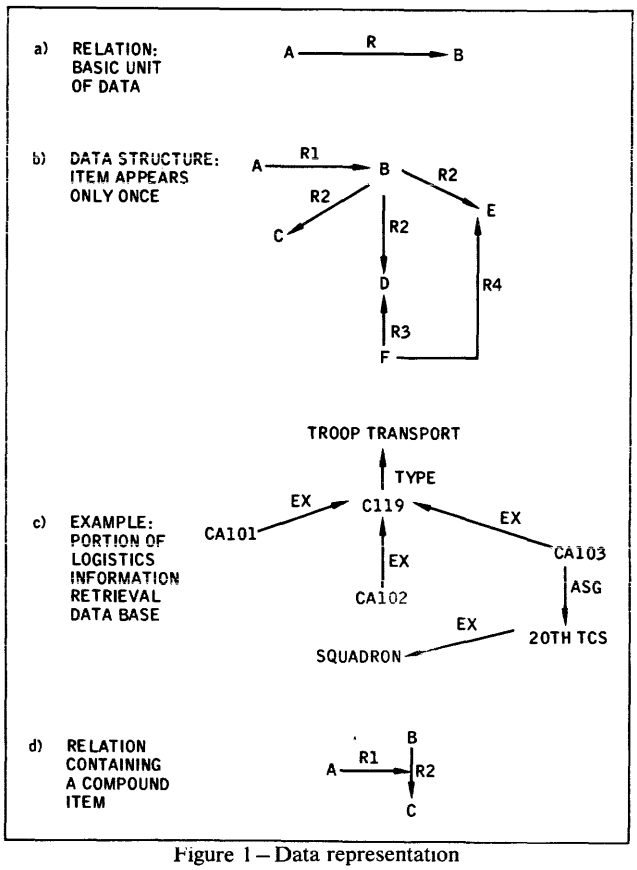


Figure 1 – Data representation

Because the associations between data items are explicitly stated in the data relations, the meaning of a data structure is generally apparent from an examination of the data alone. This may be illustrated by reference to Figure 1c, which shows the directed graph representation of a portion of a logistics information retrieval data base. The link labels *ex, asg,* and *type* are abbreviations for the relationships "is an example of," "is assigned to," and "is a type of," respectively. The directed graph therefore conveys the following information:

C119 is a type of TROOP TRANSPORT
CA101 is an example of a C119
CA102 is an example of a C119
CA103 is an example of a C119
CA103 is assigned to the 20TH TCS
20TH TCS is an example of SQUADRON.

*As an example, a translator has been programmed in the ASP language for translating queries expressed in a large subset of the OTC query language (developed for the Air Force 473L data processing system) into single ASP instructions. This translator consists of 23 instructions; the translation process, for a moderately complicated query, required 39 instruction executions.

An item in a relation may itself be an association between two items, and have the form of a relation. Such an item is referred to as a "compound item." In the directed graph format, a compound item is placed into a relation by connecting one end of the relation's link to the center of the link in the compound item, as illustrated in Figure 1d. Compound items may themselves have compound items as items, to any depth. As with simple items, a compound item can be connected to any number of other simple or compound items.

The ASP instruction

The ASP language instruction is a specification of a conditional transformation to be performed on the data. The transformation and the condition upon which it depends may be very elaborate.†

A relatively short program of these instructions can specify any process to be performed on the data base, including those involving arithmetic, string and input-output operations.

Specifically, the ASP instruction commands that the data base be searched for matches to one set of relations and, if the search is a success, that the located data be replaced by data specified by a second set of relations. In addition, each instruction specifies the instruction to be executed next as a function of the search result (success or failure), and thereby provides the conditional branch capability needed for programs.

ASP instructions themselves are expressed as structures of relations, and are stored with the data. These structures are linked together to form programs. A special item and several special link labels are reserved for use in identifying instructions, and in distinguishing their major components. Because there is a common representation for data and instructions, one ASP program can be processed by another ASP program.

For programming convenience, an instruction format is used which differs somewhat from the representation just mentioned. Because it is also a more convenient format for discussing instruction interpretation, it is used in the present paper. This format, shown in Figure 2a, consists of four fields. The *control structure* field specifies the sets of relations to be located in the data base. The *replacement structure* field specifies the sets of relations which are to replace the relations located in the data base. To separate these two variable-sized fields, a bold arrow is placed between them. The instruction is labeled in the *name* field. The names of the instructions which can be executed next are specified in the *go-to* field. The name of the next instruction for the search success case is preceded by an S. The name of the next instruction for the search failure case is preceded by an F.

The control structure

The control structure of an ASP instruction may contain any number of relations. Each item and link label in one of these relations is either a known data word, or one of the special symbols, *X* and *Y*. These special symbols are referred to as "variables," and are used to specify *unknown* items or link labels. The control structure in Figure 2a, for example, contains two relations with a common *X* variable.

The execution of an ASP instruction begins with a search for all sets of data relations which match the set contained in the control structure. For each relation in the control structure there must be a matching data relation in each data set. The criterion for a match between two relations is that each of the corresponding items and link labels must match. A pair of items or link labels are defined to be matches if they are identical or if one of them is a variable. A data item or link label which matches to a variable is referred to as a "value" for the variable. When, as in Figure 2a, two or more relations in a control structure contain a common variable, the corresponding relations in each set of matching data relations must contain a common value for this variable.

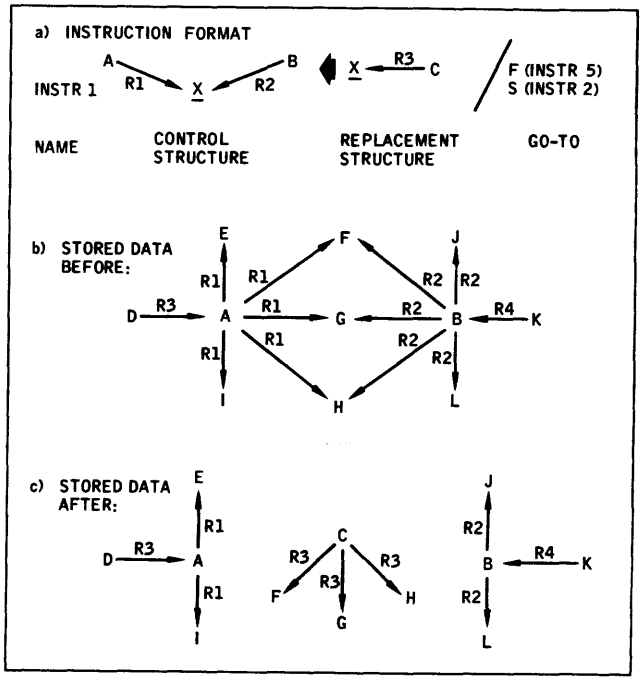


Figure 2—Example of an instruction and its operation

†The ASP instruction actually corresponds, in power and generality, more nearly to a statement in a high-level programming language than it does to an instruction in a conventional computer.

The interpretation of the control structure may be illustrated by reference to Figures 2a and 2b. Figure 2a shows an instruction, and Figure 2b shows a data structure on which the instruction is to operate. The control structure of the instruction specifies a search for all pairs of data base relations such that

(1) one relation in the pair contains an R1 link from item A to any item, and
(2) the other relation in the pair contains an R2 link from item B to that same item.

The data base shown in Figure 2b contains three sets of relations which match to this control structure. They are

(A,R1,F) and (B,R2,F),
(A,R1,G) and (B,R2,G),
(A,R1,H) and (B,R2,H).

Each of these matching sets of relations contains a value for the variable *X*. The values are F, G, and H, respectively.

It can be seen, from this example, that the control structure provides the means to retrieve values of unknown items or link labels from the data base by specifying the context of relations in which the unknown appears. This feature of the ASP language, which may be referred to as "context addressing", permits data to be retrieved by association. It may be contrasted to retrieval by memory location address, which is the basic retrieval criterion for conventional digital computers. The specification is basically a Boolean AND function of relations involving the unknown. However, any complex Boolean function can be specified by using the special link labels AND, NOT and OR. The details are not discussed here.

The *X* is used as the variable symbol when it is desired to specify the retrieval of the set of *all* values which can be located in the data. The *Y* symbol is used to retrieve just *one* value of the variable. The particular value selected for a *Y* is arbitrary as far as the programmer is concerned. (Any machine for implementing the ASP language would probably be deterministic in this respect; however, the language is not.) If a *Y* had been used in the example above, its value would have been one of the three items, F, G, or H.

A control structure can contain any number of relations involving any number of variables, and interrelated in any fashion. A particular relation may contain more than one variable. This permits the specification of one unknown in terms of one or more other unknowns. Subscripts may be used to distinguish the variables when more than one is used in an instruction. Examples of complex structures are given later in the paper.

As previously indicated, the result of the control structure matching operation determines the sequence of events in the execution of an ASP instruction. If at least one set of data relations can be matched to the control structure, the instruction execution is said to be a "success." In this case the data modification specified by the instruction will be performed next, and control will then be passed to the instruction which is specified with an S prefix in the go-to field. Otherwise the matching operation is a "failure," and control is immediately passed to the instruction specified, with a F prefix, in the go-to field.

Replacement structure

The replacement structure also may contain any number of relations, involving both known and variable items and link labels. It is required that any variable which appears in the replacement structure must also appear in the control structure as well.

If the matching of an instruction control structure is a success, the instruction execution concludes by replacing all of the data relations which matched to the control structure with the relations specified by the replacement structure. A replacement structure relation which contains a variable, indicated by the *X* symbol, is interpreted as a set of relations to be stored. There is one relation for each value of the variable which was identified when the control structure was processed.

The interpretation of the replacement structure may be illustrated by the example in Figure 2. The replacement structure shown there has one relation which contains the variable item whose values were determined when the control structure was processed. The values of this variable were determined from the data base in Figure 2b to be F, G, and H. This replacement structure, therefore, specifies that the three pairs of relations located by the control structure (see above) be replaced by the three single relations

(C,R3,R)
(C,R3,G)
(C,R3,H).

The resultant modification in the data base can be seen by comparing the stored data base shown in the "after" picture (Figure 2c) with the data base shown in the "before" picture (Figure 2b).

Summary of instruction interpretation

The control structure specifies sets of relations to be located in the data base, and the replacement structure specifies sets of relations which are to replace the relations located in the data base. The difference

between the specification of relations to be located and the specification of relations to be stored is itself a specification of a modification to the data base. An instruction with a blank control structure will effect a simple store operation, since it only creates the new relations specified in the replacement structure. An instruction with a blank replacement structure will effect a simple deletion operation, since the data relations specified by the control structure are replaced with nothing.

Matching data by implication

The interpretation of the control structure specification, which is given above, refers only to the direct match feature of the ASP concept. Perhaps the most striking feature of the ASP concept is the capability that it provides for matching, indirectly, to data which may be inferred to be in the data base on the basis of statements of implication stored with the data base.

The capability for matching data indirectly in this fashion makes it possible for the control structure of an instruction to match data relations when the data base and control structure convey the same associations, but in different levels of detail or in equivalent, but not identical, formats. For example, this capability allows indirect matches to be made between a set of stored data relations and a single control structure relation on the basis of the data relation link label having transitive properties. If the data base included the relations

(C119, is a type of, TROOP TRANSPORT)
(TROOP TRANSPORT, is a type of, AIRCRAFT),

it would then be possible to find an *indirect* match to the control structure relation

(C119, is a type of, AIRCRAFT),

provided that the transitive rule of inference for the relationship "is a type of" is specified by a statement of implication stored in the data base.

The statements of implication are themselves expressed as relations of the same form as that used to represent data. The format employed is not discussed here, except to say that it involves the use of compound items. Since the statements of implication can be stored and manipulated in the same way as any other data, they may be regarded as statements of generality, as other relations are regarded as statements of particulars.

Generated data

In any practical device for interpreting the ASP language, there would be a number of types of relationships for which, because of sheer quantity, the explicit and implicit retrieval techniques discussed thus far would clearly be impractical. The arithmetic and quantitative relationships on pairs of numbers are good examples.

In order that the matching concept of the ASP language apply to such relationships, the associated relations must be generated by the hardware of the device, or by a subroutine, when a control structure specifies that they be located in the data base. A relation which is generated by hardware must employ a reserved-meaning item or link label (which corresponds in function somewhat to the operation code of an instruction in a conventional computer) in order to initiate the generation process. A relation which is generated by a subroutine must employ an item or link label which a programmer has previously specified as the name of this subroutine.

Typical of the relations which are generated in the ASP concept are SUM, DIFFERENCE, COUNT (the "count" of the data items which match to a control structure item), and GREATER THAN. Two of the most important types of relations which are generated are DECONC, which relates a simple item to a sequence of its component characters, and CONC, which relates a sequence of simple items to the single item which is their concatenation. These latter two relation types make it possible to apply the full power of the ASP concept to problems involving string manipulations.

Input and output

Input and output operations are specified by using relations with the special link labels INPUT and OUTPUT in the replacement structure. The values of variables, or the matching data relations located in the data base by the control structure, can be output by an OUTPUT relation in the replacement structure of that same instruction. (An example is given in the next section.)

Specifying an input device in an INPUT relation will cause relations to be read in from that device and stored in the data base.

Programming examples

Most complex queries and data modifications can be specified in a single ASP instruction. Furthermore, there is a very close structural similarity between such an ASP instruction and an equivalent query or modification statement as it would be expressed in an English-like query/maintenance language. This suggests that the task of translating from such a language to the ASP language should be relatively straightforward.

An example of a powerful one-instruction ASP program and its English-like equivalent is shown in Figure 3. This ASP instruction specifies a modification in a

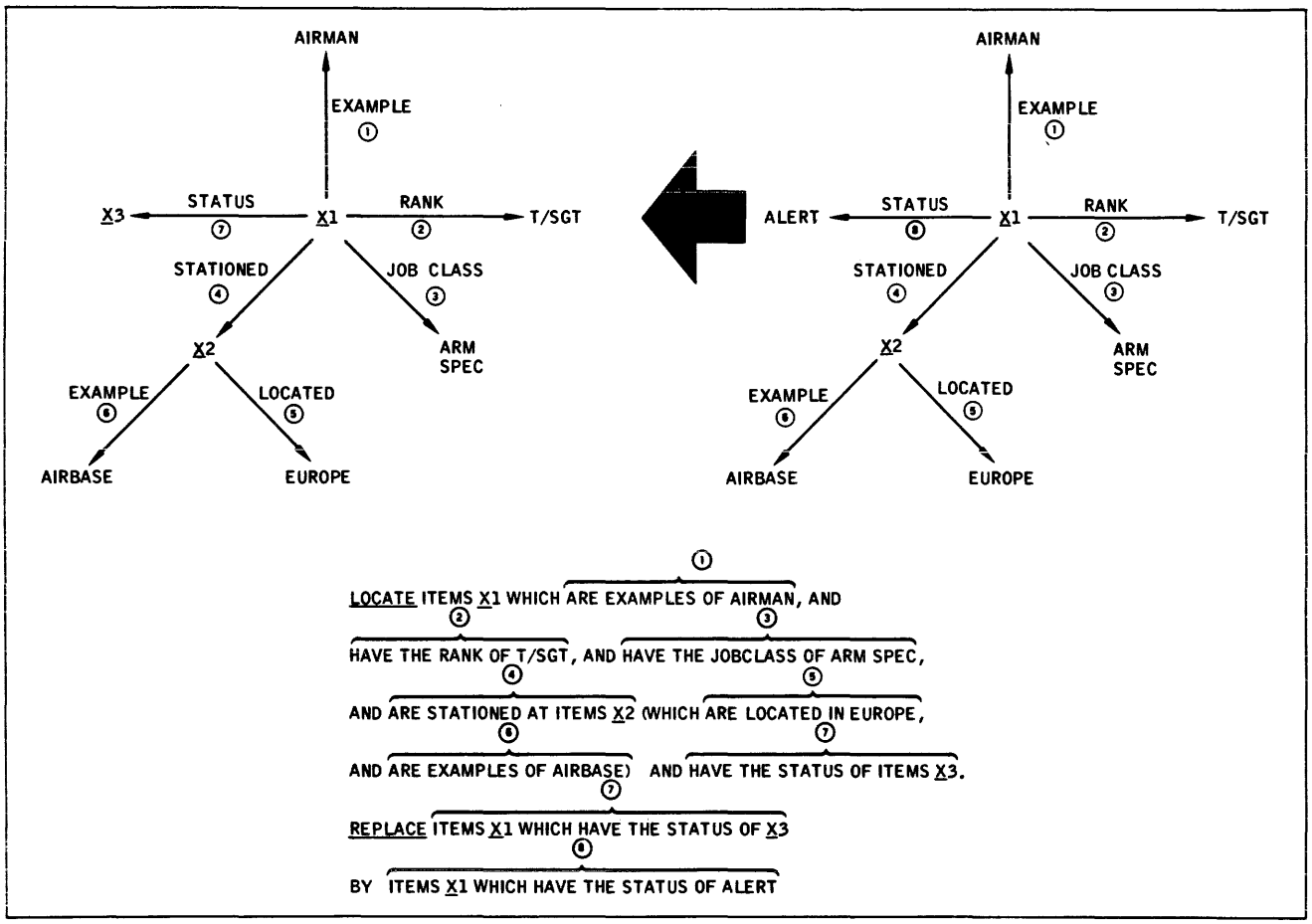


Figure 3 – Example of a complex ASP instruction

hypothetical military information retrieval system. The specific modification is in the status of all examples of airmen ($X1$) which meet several criteria, including that they be stationed at airbases located in Europe. Whatever their status ($X3$), was previous to execution of this instruction, it is changed to ALERT. The correspondence between ASP relations and phrases in the sentences is shown by the circled numbers.

Note that the only difference between the control and replacement structures is in the "status" relation. The other relations are required to select the desired values of $X1$, but there is no intention, in this example, to alter them. Note also that it would be simple to augment this same instruction to specify the retrieval of information, as for example, the names of the affected airmen. This would be accomplished by adding the relation ($X1$, OUTPUT, TYPEWRITER) in the replacement structure.

An ASP program can be written to perform any process which can be described by an algorithm. ASP programs may include arithmetic operations, character manipulations, list processing, etc. Some of these operations, which are called out by reserved-meaning ASP symbols, involve indirect matches. Several of these symbols are introduced and explained in the example program which is discussed next.

Before discussing the program example it should be mentioned, explicitly, that variables are defined only within individual instructions. The values of variables are determined when the control structure containing them is processed, and new relations can by created with these variables only by the replacement structure of that same instruction. The values of variables appearing in another instruction are determined with no reference to variable values of other instructions, when that instruction's control structure is processed.

Figure 4 shows an ASP program for performing a complex search on data and outputting the results. The function of this program is to retrieve the names of all documents, published in the U.S.A. after 1958, whose principal topic is ARTIFICIAL INTELLIGENCE, and which have as descriptors at least three of the following: HEURISTIC, PROBLEM-SOLVING, THEOREM PROVING, INDUCTIVE INFERENCE, DEDUCTIVE INFERENCE.

The program begins by selecting those documents which satisfy all necessary criteria (i.e., published in U.S.A. after 1958, principal topic is ARTIFICIAL

INTELLIGENCE) and putting all of them in a particular relation (that of being an example of a "candidate"), after first deleting any other relations (*X*, example of, CANDIDATE) which may have already existed in data. All of this is accomplished in the first instruction (INSTR 1) of the program.

In addition, the first instruction creates a relation on each document which assigns to it a tally with an initial value of zero. Any old tally relations are deleted. Note the modifier "NON-TEST" which is appended to the link labels of two of the relations in the control structure of the instruction. These are the relations which identify any previous examples of candidates or values of tally which may have been left in data after a previous execution of the program. The NON-TEST modifier specifies that the pattern match shall not fail if no data relations corresponding to the relation it modifies are found.

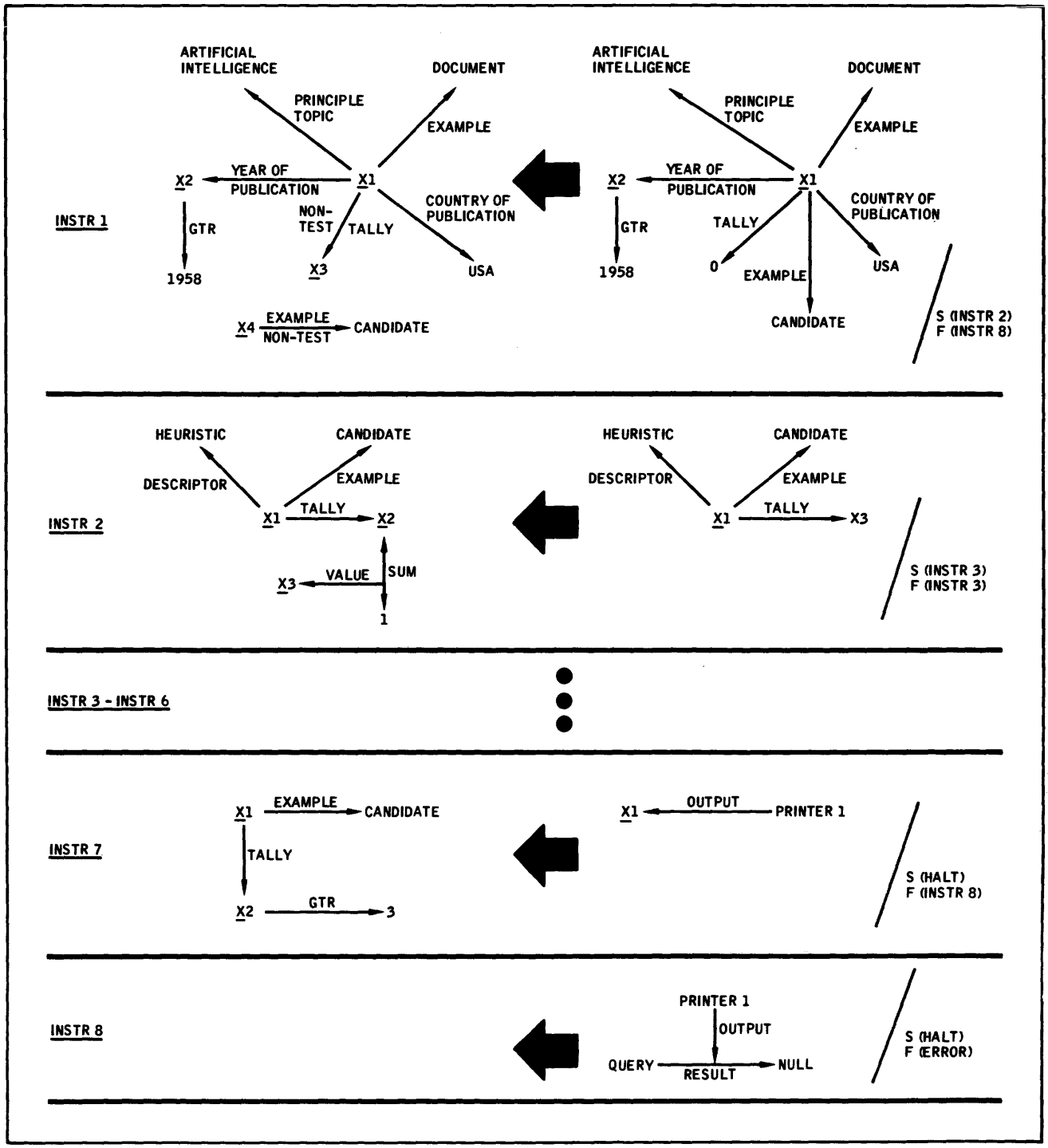


Figure 4—Example of an ASP program

The second instruction locates all of those documents which, in addition to satisfying the requirements of the first instruction (they are "candidates"), have HEURISTIC as a descriptor. The instruction also locates the "value" of the tally for each such document and locates (indirectly, by virtue of the indirect match involving the reserved symbol SUM) the sum of one plus the value of the tally. The replacement structure then specifies, in effect, that this incremented tally value replace the previous tally value for all documents located by the control structure. The syntax in which SUM is used is fixed, so that the hardware of a processor will have predefined internal "locations" from which to fetch the numbers to be added, and a predefined location for inserting the resulting sum.

The third through the sixth instructions, which are denoted only by the three dots following the second instruction, are like the second, except that the item HEURISTIC is replaced, successively, by the items PROBLEM-SOLVING, THEOREM-PROVING, INDUCTIVE INFERENCE, and DEDUCTIVE INFERENCE.

After the execution of the sixth instruction, those documents which satisfy at least three of the five criteria given in the five previous instructions will have a tally of 3 or greater. The second to last instruction, INSTR 7, locates the documents satisfying this criteria (those having a tally value which is greater or equal to 3), and outputs their names to a device called PRINTER 1. Two reserved-meaning symbols, GTR (for "greater than") and OUTPUT appear in that instruction. The context of GTR is self-explanatory. The output function, specified by the appearance of OUTPUT in the replacement structure, causes the output (to PRINTER 1) of all values of $X1$.

The final instruction is the output instruction in case of the failure of any document to have a tally of 3 or greater (in which case there are no documents satisfying the search criteria). It outputs the relation (QUERY, result, NULL), in unformatted string form, as an ordered triple. If more elaborate formats are desired, the output function may be specified by a special subroutine, rather than by the instruction or program defining the query.

PART II ASP MACHINE

The ASP machine organization described in this part of the paper is radically different from that of a conventional digital computer. This is a result of the fact that it was designed specifically to implement the ASP language. This ASP machine organization permits the entire data base to be processed in parallel by the ASP instruction. As a result, the instruction execution time is dependent almost solely on the complexity of the control and replacement structures, but is nearly independent of the amount of data to be searched or altered by the instruction.

The ASP machine organization achieves its high speed, parallel processing capability by taking advantage of the low cost of logic implemented in large scale integrated (LSI) circuitry. The LSI circuitry is used to implement an iterative-cell, distributed logic memory. This memory provides the parallel search features of an associative memory,[10,11] plus an extremely powerful inter-cell communication feature. It is this latter feature which enables the ASP machine to parallel-process the entire data base for the more complex operations involved in the instruction execution.

The key element in achieving high-speed execution of the ASP instruction is the matching of the control structure relations to the data base. These relations may be divided into two catagories, relations which can be matched independently of each other and relations which are interrelated.

A control structure relation which contains no variable (X or Y) which is common to another relation can be matched independently. With the data base stored in an associative memory, it can be quickly searched in parallel for all matching relations.

However, if (as in Figure 2a) there is a variable which is common to a set of two or more control structure relations, these relations cannot be matched independently. Matching sets of data relations with common values for this variable must be found. This process can be quite time consuming, even with an associative memory, because of the need to check the individual matches to each relation for this common value condition. To perform this function in a highly parallel fashion, a new memory function called "context-addressing" was implemented by adding logic to the word cells of an associative memory structure.

The "context-address" of a variable is defined as the set of control structure relations which contain this variable, and thereby specify the context of data relations in which values for the variable must be found. The context-addressing memory function permits the memory to be searched, in turn, with each relation in the context address, and automatically tags the item/link label values which meet the criterion of being in each relation. As long as at least one variable value is found, the relations in the context address are known to have at least one match in the data base.

The context-address logic permits the data base to be searched for matches to these interrelated relations without having to check the individual matches

to each relation. This check is automatically accomplished within the memory by simultaneous communication between pairs of word cells. Because of the significance of this function, the memory of the ASP machine has been given the name "context-addressed memory."

With the intercell communication capability provided by the context-address logic, a relatively small amount of additional logic permits the parallel implementation of two other significant operations in the execution of an ASP instruction. One of these operations is the matching of control structure relations which involve two variables. The other operation is the writing of the set of relations specified by a replacement structure relation involving a variable.

General machine organization

The ASP machine organization is shown in Figure 5. The dominant element in this organization is the context-addressed memory. This memory stores both ASP data and programs, and, as suggested by its name, provides the capability to identify, in parallel, unknown items (and link labels) by specifying the context of relations in which the unknowns appear. This memory, its design, and the functions it performs are discussed in detail in the following sections.

Associated with the context-addressed memory are a compare register and a mask register. These registers perform functions which are similar to the functions performed by such registers in an associative memory. The compare register holds the word used to content search the memory, and also serves as a memory data register. The mask register holds the word used to mask specified bit positions so that they will not be considered in a content search of the memory.

A relatively small read-only memory is employed to store a micro-program for executing the ASP

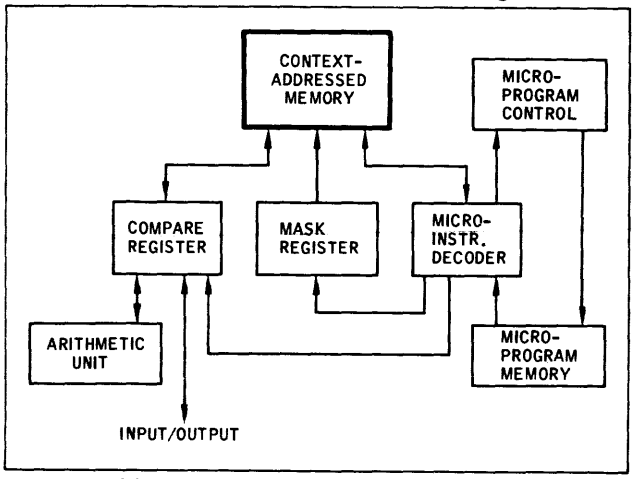


Figure 5 – An ASP machine organization

instructions. The micro-instructions include eight functions which are performed by the context-addressed memory (e.g., search, read, context address, etc.), plus several control functions which are performed external to the memory. The micro-program retrieves the components of the current ASP instruction from the context-addressed memory, executes this instruction on the data stored in that same memory, selects the next instruction to be executed, and then recycles and repeats the process. The one micro-program executes all ASP instructions, regardless of their complexity.

An arithmetic unit performs the wired-in functions (e.g., SUM, etc.) when the micro-program encounters relations in an ASP instruction which call out these functions. The complexity of this unit depends upon the complexity of the wired-in functions desired for a particular application. It could be as simple as an accumulator, or as complex as a stored-program digital computer. It is also possible to include basic parallel arithmetic capabilities in the cells of the context-addressed memory, using techniques which have been developed for associative memories, and thereby eliminate the external arithmetic unit entirely.

Context addressed memory

The context-addressed memory consists of a square array of identical storage cells which are interconnected both globally and locally. Each cell contains both memory and logic circuitry. The memory circuitry stores either an item, link label, or a relation, plus tag bits. The main purpose of the logic circuitry is to perform the comparison operations which are required to implement global searches of the array and local inter-cell communication.

The two types of cell interconnections are diagrammed in Figure 6. The contents of the compare and mask registers and the output of the micro-instruction decoder are distributed to all cells on the global busses. The signals involved in the inter-cell communication are propagated from cell-to-neighboring cell via the local lines. As indicated in the Figure 6b, each cell can propagate a signal only to its immediate neighbors to the north and to the west. The cell at the top of each column is connected to the cell at the bottom of that column, and the cell at the west end of each row is connected to the cell at the east end of that row. As a result, any cell can communicate, through a chain of intermediate cells, to any other cell in the array.

Cell memory

The memory of the storage cell is divided into five equal-length fields. Three of these fields are used to store ASP data and instruction information, and the

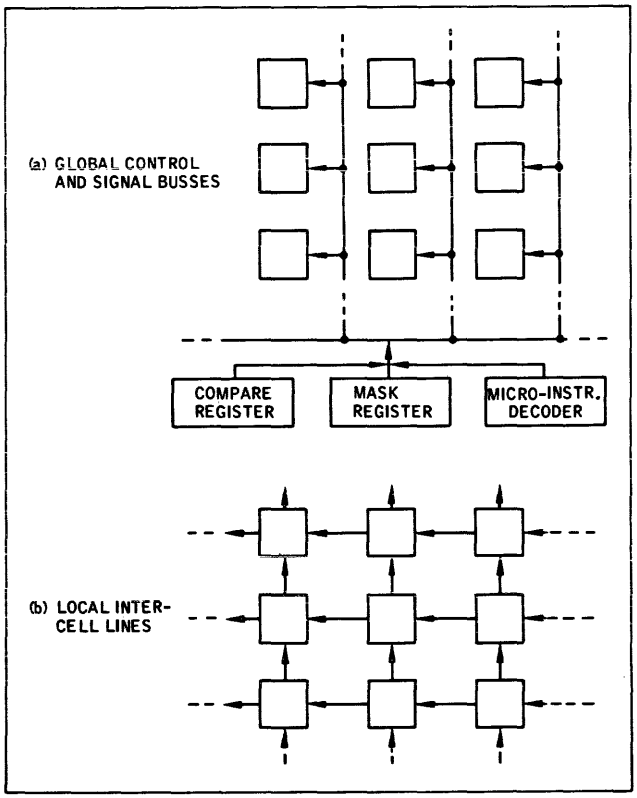


Figure 6 – Storage cell interconnections

other two fields are utilized for tags by the microprogram while it is executing an ASP instruction. The three information fields may be used to store the literal value of an item or link label, or to store a coded representation of a relation or compound item. In addition to these five fields, the cell has its own address permanently wired in to it when it is attached to the array substrate.

Data relations and compound items are coded in terms of the addresses of the cells which contain their component items and link labels. The correspondence between the directed graph and the coded representation used in the storage cell can best be explained with the aid of an example. Figure 7 shows a portion of a memory array and a corresponding directed graph.

Each square in the array diagram represents a storage cell. The data stored within each cell is shown within the corresponding square. The star (☆) symbol is used to indicate that a cell stores a relation (rather than a compound item).

The numbers along the periphery of the array represent the row and column addresses of the storage cells. A cell's address is specified by its row number, followed by its column number; for example, "CS" is stored in cell 3, 1. To simplify the figure, a cell's address is also referred to by showing the data stored in that cell, enclosed with brackets.

The directed graph shown below the memory array represents the same data as is shown stored in the array. As an example of the correspondence between representations, consider the relation

(A, R1, C).

This relation is said to be stored in cell 1, 2. The actual contents of this cell are:

A = 1,1 = address of cell containing "A"
R1 = 1,4 = address of cell containing "R1"
C = 4,2 = address of cell containing "C".

As another example, the CS relation between *I* and the compound item is

(A, R1, *X*)

is stored in cell 4,1.

Cell logic

The cell logic performs comparison operations, and generates and propagates the inter-cell communication signals. In addition, it records the results of a comparison and performs Boolean operations on the results of successive comparisons.

Comparison operations are performed by a single comparison network. This network is used to compare the global input signal to the contents of any one of the cell's five memory fields or to the cell's wired-in address. The same network is also used to compare the local input signal to the cell's wired-in address.

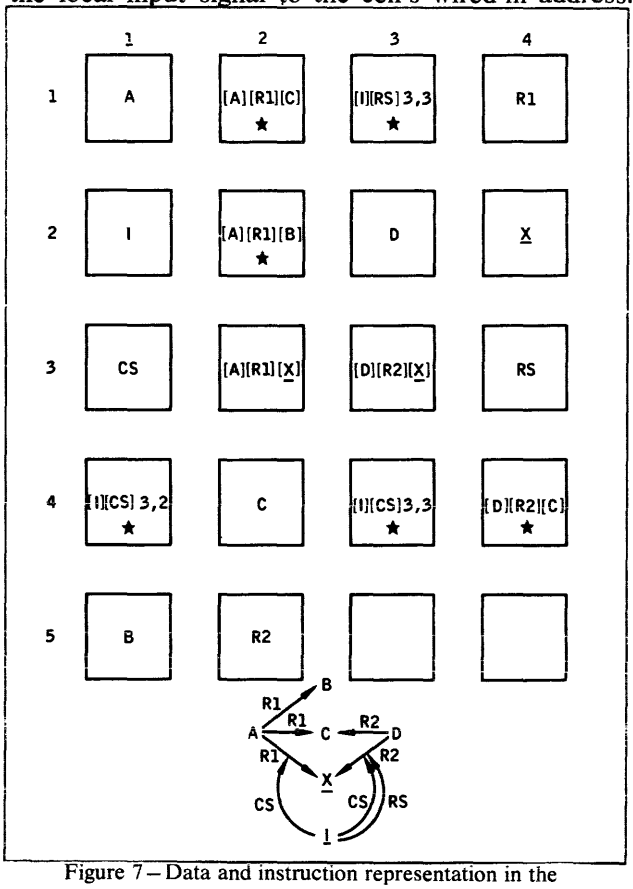


Figure 7 – Data and instruction representation in the memory array

The result of a comparison operation performed within the cell is recorded in a match flip-flop. A Boolean function of successive comparison operations can be recorded in a second flip-flop, called the "sequence" flip-flop. The Boolean function is obtained by taking a logical combination of the sequence and match flip-flops, and storing the result in the sequence flip-flop.

The initial generation and the propagation of inter-cell communication signals is accomplished by a pair of switching networks. One network can gate a signal which originates within the cell, or which enters the cell on the local lines (either from the south or from the east) out on to the local lines to the north. The other network can gate a signal which enters the cell on the local lines from the east out on to the local lines to the west. In addition to the switching networks, a pair of buffer registers are required to temporarily store the signals entering a cell on the input local lines.

Inter-cell communications

The local signals are addresses of cells. Signals are initially generated as a result of a global command which is responded to by all cells which have a particular tag set. These cells simultaneously output any one of their three stored addresses (assuming they store relations or compound items) or their own address, as specified in the command, on their inter-cell lines to the north.

The address signals then propagate from cell to cell in synchronism with a global clock, until they reach a cell with a matching wired-in address. A signal always travels north initially until it reaches the row which it specifies. It then turns left and travels west along that row until it reaches the cell which it specifies.

The direction of propagation is determined by the comparisons performed within the storage cells. A cell compares the row components of an incoming north-bound signal with its own address. If there is a match, the cell outputs the address signal to its neighbor to the west; otherwise the address signal is propagated to its neighbor to the north. A cell compares the column components of an incoming west-bound signal and its own address. If there is a match, the signal sets a flip-flop; otherwise the signal is propagated to the neighbor on the west.

Storage cells are designed to simultaneously process address signals going south-to-north and east-to-west. However, if a cell simultaneously receives an east-to-west signal and a signal from the south which wants to turn west, a conflict, called a "blockage," occurs. The cell cannot transmit two signals to the west simultaneously. The signal coming from the east is given priority and is sent on to the west. The signal from the south is blocked from entering the row and is transmitted to the north.

This blocked signal will then automatically travel around the entire array column and return to the row it wishes to enter. With the introduction of routing cells and the express routes which they control (discussed in a later section), this signal blockage does not cause excessive delay in the communication process. When a high percentage of the cells transmit signals, it is possible to have a signal blocked more than once. Preliminary analysis indicates that even in such cases the delay is not excessive.

The memory functions

The distributed logic and memory provided by the storage cells is designed to implement eight powerful memory functions. These functions (plus several executed external to the memory) enable the microprogram to execute ASP instructions of any complexity upon large quantities of data within the memory.

Each of the memory functions manipulates sets of storage cells. This set manipulation characteristic is suggested by the nature of the ASP instruction. The control structure can specify the selection of a set of values for a variable (X) which is defined by a conjunction of relations, and the replacement structure can specify the creation of a relation for each value in the set. Since the number of values for a variable may be quite large, it is imperative to parallel process the set of values without having to retrieve them from the memory.

The memory functions specify sets of storage cells either by the data or the tags in a cell memory, or by the state of a "match flip-flop" in the cell logic. Data and tags are written into cell memories by the *write* function. A set of dissimilar data relations can be simultaneously written into a set of cells, in many cases, by a *mass write* function.

The match flip-flop can be set in cells selected by the *search, context address,* or *box-car* functions,, or by any Boolean combination of these functions. The *pulse* function is used to perform the Boolean selection. The match flip-flop can be reset in all cells by the *reset* function.

The remainder of this section discusses each of the eight functions. Reference is made to Figure 8, which diagrams the signal flows involved in six of these functions. The next section gives a detailed example of the context-address function.

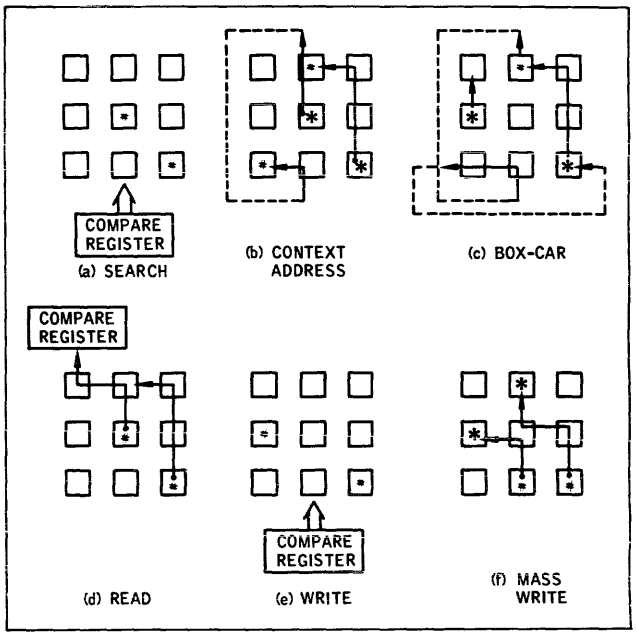


Figure 8—The principal memory functions

Search

The search function is identical to the content-addressing function implemented in conventional associative memories. This function selects, in parallel, all cells whose contents match the contents of the compare register, in the field specified by the mask register. As indicated in Figure 8a, this function employs the global lines and no local lines. The match flip-flop is set in all matching cells (indicated by # in the diagram).

Context address

This function selects, in parallel, all cells which are specified in a particular data field of a set of transmitting cells. The main use of this function is to select tentative values of variables in the pattern matching of control structure relations. The set of transmitting cells contain matches to a control structure relation, and the selected set of cells contain tentative values for the variable in this relation.

The context address function is always preceded by operations which select the transmit cells (usually a search) and set their match flip-flops. A write function is then performed on these cells to set the transmit tag bit and to write into two bits which specify the field to be involved in the context addressing.

When the context address function is called out, all cells whose transmit tags are set respond by transmitting the address contained in the specified field to their neighbors to the north. Each of these address signals travels on local lines to the cell which it specifies, and sets the match flip-flop in that cell. Figure 8b shows two cells whose transmit tags are set (indicated by *) transmitting signals on local lines to two other cells (indicated by #).

Box-car

This function selects, in parallel, all members of a transmitting set of cells which specify, in a particular data field, members of a second set of cells. The main use of this function is in the pattern matching of control structure relations which contain two variables. After the sets of tentative values have been obtained for these two variables (using the context address function), the box-car function is employed to select the set of cells containing relations involving tentative values of each of the variables. (This is explained below with an example.)

The box-car function is preceded by operations which select the set of transmitting cells, set their transmit tag, and store the specification of the field to be transmitted.

When the box-car function is called out, all cells whose transmit tags are set respond by transmitting the address contained in the specified field to their neighbors to the north. In this respect the function is similar to the context address function. However, in this case, as the address signal is initially transmitted from each cell, the address of the transmitting cell is "attached" to it.

Refer to the address contained in the designated field as "A" and to the address of the transmitting cell as "B." As A travels from cell to cell, first north and then west (until it reaches the cell whose address is "A"), B follows it, somewhat as a box-car following a locomotive.

The A signal finally reaches the cell whose address is "A." The fate of the B signal, which is attached to the A signal, depends upon the state of the "box-car" tag in the cell. If the tag is reset (indicating that the cell had not been selected on the search which preceded this box-car operation), the B signal is destroyed. However, if the tag is set, the cell will retransmit the incoming B signal. The B signal will then travel from cell to cell until it returns to the cell whose address is "B." When it reaches that cell it sets the cell's match flip-flop.

The diagram of Figure 8c indicates typical signal flows for the box-car operation. Each of a set of relation cells whose transmit tags are set (marked with #) simultaneously transmits, via local lines, to a cell in another set. If a receiving cell has its box-car tag set (indicated by a * for one of the two receiving cells) the box-carred address is retransmitted. This retransmitted signal must travel across the array, back to the cell which initiated it. In the diagram only one of the original transmitting cells receives back a retransmitted signal.

As an example of the application of the box-car function, assume that the data base is being searched

for matches to a control structure containing the relation ($X1$, R1, $X2$). Tentative values are found for $X1$ and then for $X2$ by performing context-address functions with all relations involving these two variables. However, the constraint that values of $X1$ must be R1 – related to value of $X2$ is not applied with the context address function.

To apply this constraint, the box-car function is employed. The transmit tag is set in all relations containing the link label R1. The box-car tag is then set in each cell which is tagged as containing a value of $X1$. Then those cells which have their transmit tags set are commanded to transmit box-car signals with their left field address as the "locomotive." At the conclusion of the box-car operation, the only cells which have received retransmitted signals are those which contain a tentative value of $X1$ as the left item and R1 as the link label.

A second box-car operation is then performed with the relation cells selected on the first operation and with the cells tagged as containing tentative values of $X2$. At the conclusion of this box-car operation, the only cells which receive retransmitted signals are those which contain a tentative value of $X1$ as the left item, link label R1, and a tentative value of $X2$ as the right item.

A pair of context-address operations is then performed with the selected relation cells to establish the values of $X1$ and $X2$ which meet the condition of being related to each other by R1. The significance of this box-car operation is that it performs the final selection of the values of interrelated variables in a highly parallel manner, without having to process the values individually.

Read

The read function causes the contents of the compare register to be replaced by the contents of a memory cell which has its match flip-flop set. (The case in which more than one cell has its match flip-flop set is discussed below.) Only one specified field may be read out at a time.

As illustrated in Figure 8d, the read function is implemented with the local inter-cell lines. The global read command gates the north-bound local lines from storage cell 1,1 into the compare register. The read command also forces every cell whose match flip-flop is set to initiate a box-car type of operation. The "locomotive" signal for every cell is the address 1,1. The box-car signal is the field specified for read out.

When cell 1,1 receives an input it discards the locomotive signal and outputs the box-car signal, as though it were doing a box-car operation. Since the north-bound local lines from this cell have been gated to the compare register (see above), this retransmitted signal is gated directly into the compare register.

If more than one storage cell has its match flip-flop set previous to a read operation, the micro-program may either:

(a) accept only the first cell's contents which reach the compare register, by terminating the read command at this point, or

(b) accept a stream of cell contents, passing the data through the compare register to a peripheral device.

In some cases, the micro-program must determine merely if at least one match flip-flop is set in the memory. A read function is then called out, but the micro-program only senses whether something has entered the compare register.

Write

This function writes the contents of a field of the compare register into the corresponding field of all cells whose match flip-flop is set. As indicated in Figure 8e, the compare register communicates to the cells via the global busses. The field is specified by the mask register. This function is used to write new data, and to set and reset the various tag bits (including the transmit and box-car tags). It is also used to write the field specification involved in the context address, box-car, and read functions.

Mass write

This function simultaneously writes the addresses of one set of cells into another set of cells which are tagged as being available for storing new data. The function is used when storing the set of relations specified by a replacement structure relation containing an X variable.

When the mass write function is called out by the micro-program, every cell which has its match flip-flop set transmits its own address. Each of these signals travels on local lines to the closest cell which is available for storing new data. When a signal reaches an available cell it is written into all three data fields. (This is a very powerful operation, as it simultaneously writes many unique data fields into unique cells.)

The implementation requires that the available tag of each cell (a tag bit which, when set, means the cell is available to store new data) be OR'ed with the available tag of every other cell in the same row, to form an "available line."

The mass write function is used when it is desired to write a set of relations such as (value of X_k, R_1, A). Previous to the mass write operation, every cell

containing a value of X_k is tagged. At entry into the mass write function, the tagged item cells send their addresses north. Each signal proceeds north until it encounters a row with a TRUE available line. It is forced to turn west here. It then proceeds along the row until it finds an available cell. The signal is then written into all three fields of this cell, the cell's available tag is reset, and its match flip-flop is set. If another signal is also on that row when the available line goes FALSE, it is forced north again, to find the next row with an available cell.

Note that for this instruction the comparison function of the storage cells is disabled, and the only controlling factor on a signal is the state of the available lines.

The diagram of Figure 8f indicates typical signal flows for the mass write operation. The two cells marked with a # symbol have been tagged to transmit one of their own addresses. The two cells marked with an * symbol have their available tag set. Both transmitted signals begin to travel towards the available cell in the middle row. However, after one signal reaches this cell the other signal turns north again, traveling towards an available cell in the next row.

The other two data fields are then written by a normal write function into all of the cells whose match flip-flops have been set. In the example above, R1 and A are written into the link label and right item fields of these cells.

Reset

This function resets the match flip-flop in every cell. It is used, at the beginning of almost every microprogram subroutine, to preset the memory. It also sets the "sequence" flip-flop, discussed below, in every cell.

Pulse

This function resets a "sequence" flip-flop in the cell's logic if the match flip-flop is not set, and simultaneously resets the match flip-flop. The pulse function is used to perform AND and OR functions of a sequence of states of the match flip-flop.

The most significant use of this function is during the selection of tentative values for a variable which is contained in two or more control structure relations. Here it is necessary to perform the context-address function with each relation in turn and to conclude with sequence flip-flops set in those cells which received an input signal at every context address operation. At the end of each context address operation, those cells which have been addressed will have their match flip-flop set. The pulse function AND's the match and sequence flip-flops and stores the result in the sequence flip-flop.

Example of the context address function

The implementation of the context addressing function may be explained with the aid of an example. Figure 9a shows a 6 × 5 segment of a memory array which stores the data structure shown in b of that figure. In this example, the data base will be context-addressed with the single control structure relation shown in c of the figure.

From the discussion of the ASP language it is apparent that pattern matching the relation in c to the data in b must result in selecting A, B, C, and D as the four values for the X variable.

The context addressing operation is begun by searching for matches to the relation shown in c.

The field containing the X is masked off during this search. Since relations are represented in memory by the addresses of their component link label and items, the memory must actually be reached for

link field = 5, 1
right item field = 5, 4.

Since compound items are also stored in terms of addresses, this search criterion will already be in this form when it is retrieved from memory as the link and right item fields of a compound item linked to an I item.

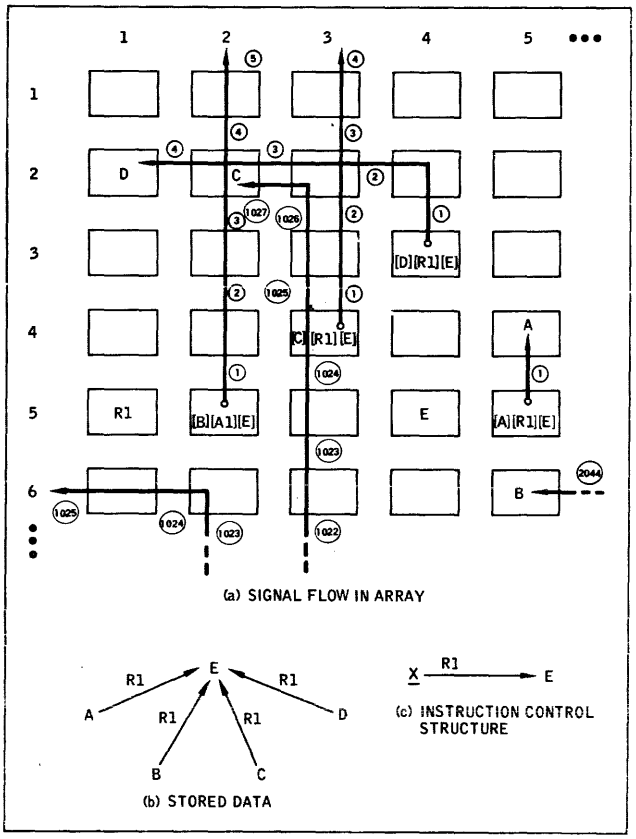


Figure 9—Example of context addressing operation in the memory array

When searched with the relation of c, four cells in the memory are found to contain matches: cells 5,5; 3,4; 4,3; and 5,2. A global write command is issued causing a transmit tag to be set in all matching cells. A two-bit tag field is simultaneously set in all of the matching cells to record the designation of the field containing the variable which is being context addressed (the left item field, in this example).

Next, the reset function is called out to tag all cells as containing tentative values of the variable, X, which is being context addressed. (All match flip-flops are reset and all sequence flip-flops are set.) The actual context address command is now issued to all cells. This global command causes each cell which has its transmit tag set to output the specified address signal to its neighbor on the north. Then the signals travel from cell to cell, on successive clock cycles, until each signal reaches the cell which it specifies. When a signal reaches the cell which it specifies it sets the match flip-flop in that cell.

The paths taken by the signals in this example have been drawn on the array in Figure 9a. The numbers shown alongside of the paths indicate the clock cycles on which the signal transfer occurs. It has been assumed here that the array segment shown in the figure is the upper left corner of a 1024 × 1024-cell array. (It has also been assumed that no routing cells, discussed below, are used in the array. With routing cells the number of clock cycles required in this example would be 100 cycles.)

Several aspects of the inter-cell communication are illustrated in this example:

(1) The signal initiated at cell 5,2 is the address of B:6,5. Since cell 6,5 is to the south-east of the initiating cell, the signal must travel north around the array and then west around the array.

(2) Two signals pass through cell 2,2 on cycle 3. No conflict occurs because one signal is going north and the other signal is going west.

(3) The signal initiated at cell 4,3 attempts to turn west in cell 2,3 on clock cycle 2, but it is blocked by the signal entering this cell at this time from the east. The blocked signal is forced to travel north around the array, to finally turn west on cycle 1027.

If the variable, which is being addressed, is contained in more than one relation, the above process is repeated for each relation, starting each time with the tentative values for the variable which have been selected up to that point and using the pulse function to accomplish the AND function of successive context address operations. It can be seen that the tentative values for the variable never have to be processed individually during the context addressing of this variable.

The number of clock cycles required to context address a variable with a single relation depends upon the greatest "distance" (number of cells) which must be traversed by any one signal and by the delay through a cell. This distance depends upon the array size, upon whether or not routing cells are used (see below), and upon the number of times that a single signal may be blocked. The number of times that a single signal is blocked depends upon the percentage of cells which are transmitting signals.

Routing cells

The storage cell which originates a signal and the storage cell which the signal specifies may be any "distance" apart, because there is no constraint on where data is stored in the memory array. For a square array of C cells, the maximum "distance" across the array is $2\sqrt{C}$ cells.

An order of magnitude reduction in this maximum can be affected by the addition of a relatively small number of a second type of cell, called a "routing" cell. The function of the routing cells is to route a signal around a block of storage cells when the specified cell is beyond that block.

The routing cells are arranged in a grid pattern, dividing the array of storage cells into square subarrays. (See Figure 10.) Each routing cell in a grid row communicates via "local" lines to the storage cell immediately above it and to the storage cell immediately below it. It communicates via "express" lines to the routing cell directly above it and below it in the adjacent grid rows. Routing cells in grid columns are similarly connected to adjacent storage and routing cells to their left and their right. These lines are indicated for one column in Figure 10.

The routing cell performs a comparison operation upon an incoming address signal to determine whether the signal should be output onto the local line or the express line. A row routing cell will output an incoming signal onto the local line only if that signal is going to a row of storage cells which are between this routing cell and the next one to the north. The column routing cells perform the analogous function with their incoming signals, using the horizontal component, or column, addresses.

A conflict may occur when a routing cell receives both a local input and an express input simultaneously. The cell can transmit only one of these input signals out on the local output line and one out on the express output line. Therefore, if both input signals request transmission on the same output line, only one request can be honored.

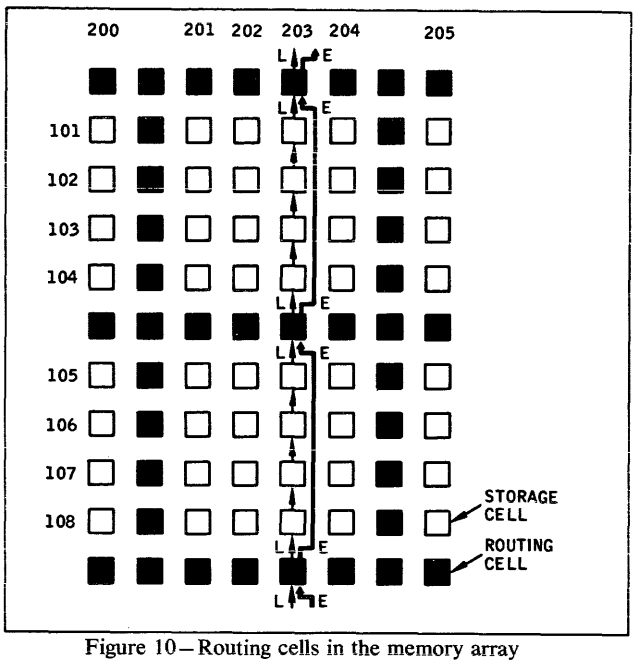


Figure 10—Routing cells in the memory array

If both inputs request to be output on the express line from the routing cell, the express input is given priority over the local input. As a result, the local input must "plod" along through another subarray of storage cells until it reaches the next routing cell.

If both inputs request to output on the local line from the routing cell, the local input is given priority. As a result, the signal on the express line is forced to stay on the express line. This signal will, as a result, travel completely around the memory column or row via the express route. It will return to the same routing cell $\sqrt{C}/s$ (where s is the subarray dimension) cycles later, at which time it will most likely be able to get on the local line.

For a memory array of 1024×1024 storage cells the optimal subarray size has been determined to be 32×32 storage cells. This reduces the maximum distance that a context addressing signal must travel (assuming no blockages), by an order of magnitude, to 186 cells. The number of routing cells required is 6 percent of the number of storage cells.

REFERENCES

1 D A SAVITT H H LOVE R E TROOP
Association-storing processor study
Technical Report No RADC-TR-66-174 AD488 538 Defense Documentation Center June 1966

2 D G BOBROW B RAPHAEL
A comparison of list processing computer languages
Comm ACM Vol 7 No 7 pp 231-240 April 1964

3 V H YNGVE
COMIT
Comm ACM Vol 6 No 3 pp 83-84 Mar 1963

4 D J FARBER ET AL
SNOBOL, a string manipulation language
Journal ACM Vol 11 No 2 pp 21-30 Jan 1964

5 P L GARVIN Technical note no 6
Inductive methods in semantic analysis
AD296428 Defense Documentation Center January 1963

6 R F SIMMONS
Answering English questions by computer: a survey
Comm ACM Vol 8 No 1 pp 53-70 Jan 1965

7 M KOCHEN ET AL
Adaptive man-machine concept-processing
AFCRL-62-397 AD288145 Defense Documentation Center June 1962

8 R F SIMMONS J F BURGER R E LONG
An approach toward answering English questions from text
AFIPS conference proceedings
Vol 2 1966 Fall Joint Computer Conference pp 357-363 Spartan Books Washington DC 1966

9 B RAPHAEL
SIR: a computer program for semantic information retrieval
AFIPS conference proceedings
Vol 26 Part 1 1964 Fall Joint Computer Conference pp 577-589 Spartan Books Washington DC 1964

10 J R KISEDA H E PETERSEN W C SEELBACH M TEIG
A magnetic associative memory
IBM Journal of Research and Development Vol 5 pp 106-121 April 1961

11 P M DAVIES
A superconductive associative memory
AFIPS conference proceedings
1962 Spring Joint Computer Vol 21 pp 79-88 The National Press Palo Alto California 1962

Digitized photographs for illustrated computer output*

by RICHARD W. CONN
Lawrence Radiation Laboratory,
University of California
Livermore, California

INTRODUCTION

Much success has been achieved in the use of computers for the control and editing of various typesetting devices.[1,2,3] At this Laboratory, similar techniques are employed in creating edited cathode-ray tube (CRT) output from information punched on Hollerith cards.[4] Many internal documents are prepared by Xeroxing the reversed film from this output at a rate of around seventy frames per minute.

It was suggested that the preparation of Laboratory catalogs could be greatly facilitated by including in the system a file of digitized photographs. The catalog text together with edit information would be punched, taped and fed to a computer together with the tape file of digitized and labeled photographs. The editing routine would call for the needed photograph by label and position it in the text in a manner specified by the edit information. Because Xeroxing at this high rate was known to produce shifts in electrostatic charge with consequent information loss in the large dark areas, it was planned to use the Xerox output as proof only and to prepare the final copy from multilith masters made from the same film.

To establish the feasibility of such a system, a series of experimental programs were written. The first few were designed to digitize film at several emulsion density levels and to record the digital pattern on magnetic tape. The next few recreated the picture in three sizes on a larger computer equipped with a high speed CRT output. The last of the series added a few words of text to the recreated photograph output.

Photograph digitizing

The large amount of filmed analog data processed at this Laboratory led to the development of a computer controlled reading device.[5] In its present state of a 5-inch CRT display, a beam splitting pellicle, and of evolution this device, called the *eyeball*, consists a pair of optical lenses, vacuum film holders, light condensing systems and photomultiplier tubes. These components are mounted on precision benches, housed in a darkroom and controlled remotely by a DEC PDP-1 computer ("Eyeball" - Film Reading System - Figure 1).

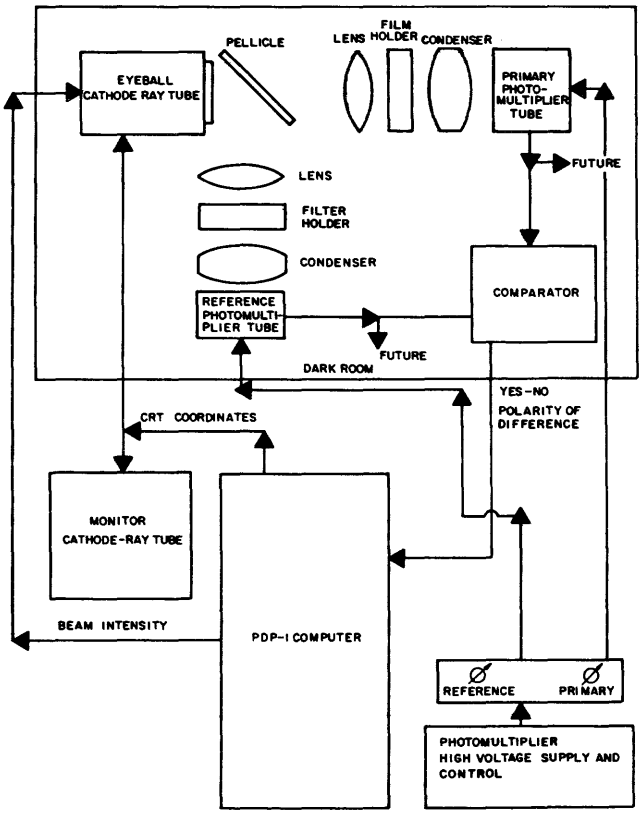


Figure 1 – "Eyeball" - film reading system

As may be seen in the figure, light from the CRT is split by the pellicle and directed toward both a primary and a reference photomultiplier tube. This light is focused against a film in the primary holder

*Work performed under the auspices of the U.S. Atomic Energy Commission.

and a neutral density filter or nothing in the reference holder. Light passing through is collected by each condenser for its respective photomultiplier.

Future plans call for the log-difference of the photomultipliers to be sampled by an analog to digital converter and made available to the computer program. It was originally planned that this scheme would be used to determine gray shades in the digitized film.

While technical difficulties with this plan are being looked into, gray scale determination is made by a programmatic variation in the beam intensity of the eyeball CRT. The photomultiplier outputs have been closely matched, so that for a single CRT beam intensity the eyeball output is a simple "yes - no," reflecting the polarity of their difference. Eyeball sensitivity is a function of the high voltage to the photomultipliers and, more importantly, of the relative voltage between the two.

The CRT coordinates as well as the beam intensity are under program control. The eyeball CRT has 4096 by 4096 addressable points over a 3-inch square. Beam intensity is variable through eight equal voltage levels. For these experiments only five could be used satisfactorily.

The latest photograph digitizing routine follows the flow indicated in Figure 2. Once the operator has positioned a film and set the photomultiplier voltage to within a known tolerance, a programmed scan sweeps the entire face of the eyeball CRT. This and subsequent eyeball operations are visible to the operator on a 16-inch monitor CRT.

During the sweep a bit pattern of eyeball yes - no responses is stored in memory. The yesses are replayed for operator observation. The operator adjusts the reference with respect to the primary voltage until he is satisfied that the yesses cover what will be the reproduction's darkest areas. (Because positives or negatives can be equally well used and can produce either positive or negative reproductions—polarity is unimportant, i.e., yes may be substituted for no, and lightest for darkest).

When the operator signals his satisfaction, the bit pattern is written as a single record on magnetic tape. The beam intensity is then reduced to the next lower level, the scan repeated, and the response pattern written as a second tape record. This sequence is repeated through the remainder of the lower beam intensities being used.

Resolution is necessarily dependent upon the mesh chosen for the programmed scan. A square array employing every sixteenth point was arrived at as a compromise between photographic detail on the one hand and time and storage on the other. It was found that the quality of the output picture was greatly improved by shifting the response pattern over and up fourth point. As on the scan, the playback pattern was shifted over and up for each subsequent record. At that time the reference coordinates were restored.

The computer time for digitizing a single film is less than a minute and can probably be reduced by a factor of two. The storage requirement is under a third of a million bits and can also be appreciably reduced by mapping out backgrounds or areas of no significance.

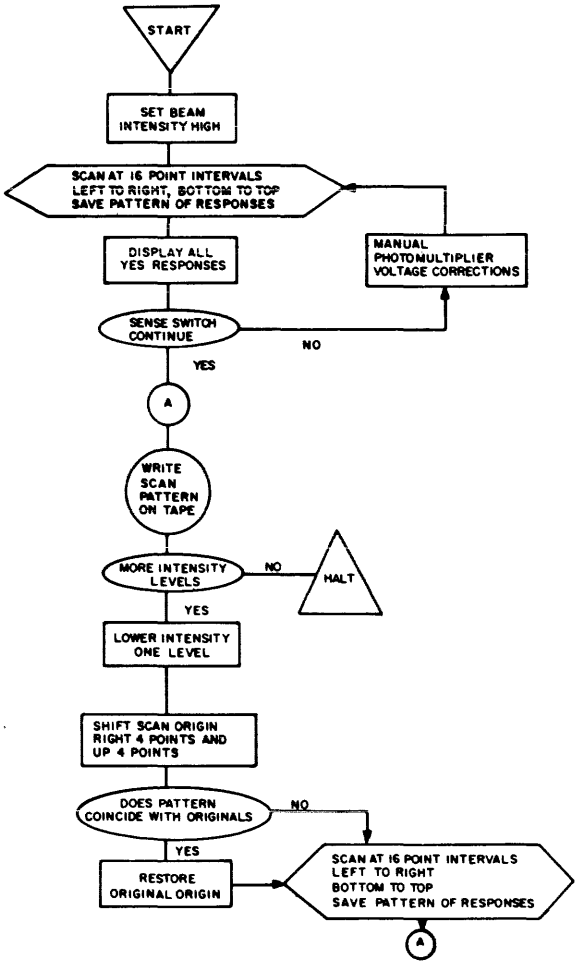


Figure 2—Flow of photograph digitizing routine

Photographic reconstruction

Although photographs were recreated for test purposes on the digitizing computer, plans for integrating the pictorial output with text made it advisable to use faster equipment with character generation capabilities included in the hardware. Accordingly, a CDC DD-80 was used for the output.[6] It is interfaced to an IBM-7094 through a direct data channel.

For film recording, the DD-80 employs a 5-inch CRT with 1024 by 1024 points across a square a little under two and one-half inches on a side. Because there are four times as many points in both directions

on the eyeball CRT, the reconstruction of film digitized at every sixteenth point is determined at every point. As on the scan, the playback pattern was shifted over and up for each subsequent record.

After commanding the plotting of a header frame and a film advance, the reconstruction program reads the digitized photograph from magnetic tape a record at a time. Following each read, all points corresponding to the pattern of yes responses are displayed. All records are displayed at a single intensity and on a single frame. The program uses less than 10 seconds of 7094 time.

Half and quarter size pictures were also recreated by displaying the response pattern with the increment between points reduced to two and one points respectively.

CONCLUSIONS

The experiments indicate the technical feasibility of including photographs with computer output. Some results are shown in Figures 4 through 6. Figure 3 is a print of the negative which was digitized for use in this paper. (The young lady is replacing the filter holder on the reference side of the eyeball.) Figure 4 is the Xerox output at full size. Figure 5 is a print made from the same DD-80 film. Figure 6 is a print made from a DD-80 film in which the text shown was included along with a half size reconstruction.

Figure 3 – Print of digitized negative

To date, all negatives have been on 4″ × 5″ cut film. Plans in progress call for the use of 35 mm roll film. This change should eliminate most of the set up time without seriously affecting resolution.

While a great deal of work is still to be done to achieve truly fine quality at low cost, the present output is satisfactory for many internal uses. It is probable that even a presentable Xerox copy may be attained by a proper prescreening of the negatives. The exaggerated flatness in shade is due largely to the use of an extremely high contrast film previously chosen for its good line reproduction characteristics.

Figure 4 – Xerox output

Figure 5 – Print of DD-80 film

ACKNOWLEDGMENTS

I would like to thank George Gielow for requesting a study of this nature, Ervie Ferris for his suggestions and help with the "Eyeball," and Sidney Fernbach for his support.

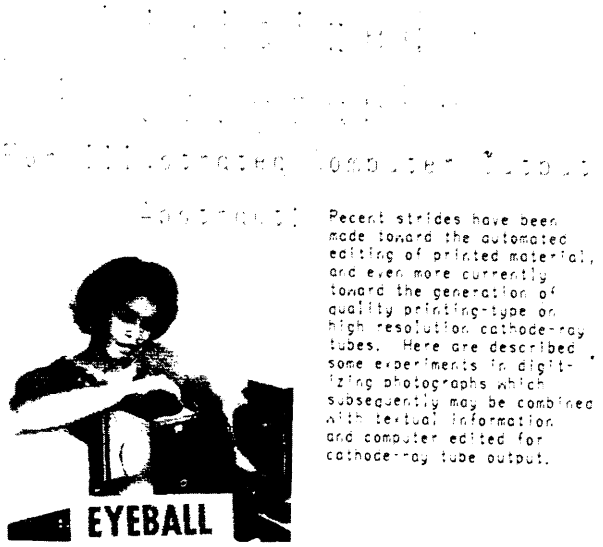


Figure 6—Print of half size reconstruction with text

REFERENCES

1 M P BARNETT D J MOSS D A LUCE K L KELLEY
Computer controlled printing
Proc 1963 Spring Joint Computer Conference Spartan Books Washington DC Vol 23 263-271 1963

2 M V MATHEWS J E MILLER
Computer editing typesetting and image generation
Proc 1965 Fall Joint Computer Conference Spartan Books Washington DC Vol 27 389-398 1965

3 R J MCQUILLAN J T LUNDY
Computer systems for recording retrieval and publication of information
Proc 1965 Spring Papers and Presentations of DECUS DECUS Maynard Mass pp 61-91 1965

4 V C SMITH
3600 publisher
CIC Report 06-001 LRL Livermore 1965

5 R W CONN R E VON HOLDT
An online display for the study of approximating functions
J ACM *12* No 3 326 1965

6 A CECIL G MICHAEL
DD80 programmers manual
LRL Livermore 1963

Mathematical techniques to improve hardware accuracy of graphic display devices

by CLOY J. WALTER
Honeywell Inc.
Waltham, Massachusetts

INTRODUCTION

The problem

Background of the problem

The contribution of scientists pioneering the design and construction of electrical apparatus has been recognized and valued by our society. Equipment ranging from toasters to air conditioners to electrocardiographs have been widely implemented and regularly used. People have viewed each new apparatus independently and sought to derive maximum benefits by judicious application of its potential.

Large scale digital computers were invented scarcely a decade ago and offered new possibilities for application of electrical design. People accepted a computer as another independent electrical apparatus, i.e., a computer was regarded as an entity. Computer peripheral devices were developed and the importance of a processing system was recognized. Although several units were coordinated to satisfy a specific requirement, no attention was given to techniques of interaction/adjustment/response among devices. Cannot a peripheral device be regarded as a governable, adjustable unit? Or a main processing unit as a sense/response mechanism that can accept/transmit information to/from elements within the system as well as control action instigated by the preassigned specific requirement?

Can a computer and a peripheral device be considered as a single unit? If a computer and a peripheral device can be considered as one unit, can the accuracy and the reliability of the peripheral device be improved by mathematical techniques? Or can improvements in electronic devices only be achieved by modifying electrical circuitry or hardware? For example, can mathematical techniques be used to improve the accuracy of a computer graphic output device despite its hardware limitations?

This developmental research problem involves discriminatory analysis, differential equations, and simulation. Methods are needed to predict the location of an error as a function of position on the screen of a Cathode Ray Tube (CRT). Compensation must be made for this error. If the errors in a CRT electron beam can be corrected before the beam is generated, then at least one of the vital questions stated earlier can be answered affirmatively. Mathematical techniques can be used to improve the accuracy of a computer graphic output device.

Reason for research

Operations performed by computer systems could be greatly expedited by an application of techniques that require an accurate computer graphic output device. Such a device is not available. The purpose of this research was to give detailed consideration to the development of a technique for converting computer digital data into accurate film image representation.

Immediate application of graphic data processing to solve a variety of problems is both physically possible and economically desirable. Graphic data processing can be effectively applied in mathematics, engineering, banking, education, communications, medicine, management information systems, programming, and many other fields. Barriers to rapid development of graphic data processing include the failure to fully develop graphic devices and to recognize benefits that can be derived from applications involving such devices.

Two recent developments make the use of computer graphic devices economically feasible. Time-sharing

enables a number of customers to utilize a single computer in a manner such that apparently simultaneous processing results. Remote data terminals permit geographically distributed installations to receive the benefits of rapid data processing as though computers were located at each facility.

Attempts have been made to improve the accuracy of computer graphic devices. New and improved circuits, equipment for precise voltage control, improved tubes, and other electro-mechanical features have been innovated. Engineers, system designers, programmers, and mathematicians recognize the desirability and ultimate necessity of providing a data processing system that accepts either digital data input or graphic input and provides accurate graphic output. Several universities and companies have developed graphic input/output devices. These devices are an advancement in the computer field but lack the degree of accuracy required for certain engineering design drawings.

The aim of this research was to contribute to the advancement of the computer and electronics fields by demonstrating that mathematical techniques can be applied to increase the accuracy of an electronic device. This goal was achieved. Such techniques were applied to General Dynamics' Stromberg-Carlson 4020 microfilm machine. Accumulated results show mathematically that a visual display system can produce accurate images, i.e., that an *inaccurate* computer peripheral device can provide *accurate* data output.

The development of an accurate computer graphic device portends significant changes. The throughput time for product design and specification and subsequent engineering modification will be greatly reduced; new vistas will open in the field of publications; new techniques will be available for the rapid production of animated films that permit the simulation of many situations. Most importantly, graphic data processing can be intimately associated with on-line systems, thus adding new dimensions to man/machine interactive systems.

Hopefully, the conceptual ideas of this paper will assist the best minds in the computer industry in a reexamination of the computer-peripheral device relationship. The result should be a faster rate of progress in the computer field than we have today.

Statement of the problem

The problem of producing an accurate image with a computer-controlled CRT was assumed to be a fundamental process control problem or a process model building problem.

Specific questions to be answered

This research was conducted to formulate bases for answers to the following questions:

1. Can a computer be used to derive equations that can predict the locations of errors in output of computer peripheral equipment? An affirmative answer to this question implies that:
   (a) Errors transmitted by an output device can be predicted.
   (b) Mathematical techniques can enable the computer to supply modified, seemingly inaccurate data to its output device.

   If these implications are valid, then an *inaccurate* peripheral device can, in turn, provide *accurate* data output.
2. Are the errors of linearity predictable in both X and Y directions? Linearity of the SC 4020 is reported as ± one percent. Prediction and subsequent correction of such variation should be possible.

Summary of related research

Many large companies and universities are engaged in computer graphic research. Interest in Engineering applications is especially prevalent. A list of aspects currently being investigated and details concerning research related to Engineering applications follows.

1. Automatic design systems
2. Programming systems and routines
3. Microfilm updating techniques
4. Improvement of accuracy by hardware advancements
5. Development of character readers
6. Console instrumentation
7. Logical arrangement of the console
8. The quality, flicker rate, modification time, drawing ability, and pointing ability of CRT units.
9. Production of hidden lines in perspective drawings
10. Development of half tones in CRT images
11. Development of color CRT images
12. Introduction of motion
13. Coupling and translation of data from one form to another form
14. Getting the structure of an image into the computer
15. Logical arrangement of drawings
16. Adequate abstraction in graphics (models are needed)
17. Engineering applications
    (a) Sandia Corporation and the Thomas Bede Foundation have jointly developed an Automated Circuit Card Etching Layout Program

(ACCEL). ACCEL was developed to design printed circuit boards and to produce drawings for their construction. The drawings are produced on an SC 4020 CRT plotter.[1] Sandia is currently trying to design and build a mosaic camera to improve the accuracy of the SC 4020.[2]

(b) Bell Telephone Laboratory is using an SC 4020 and an IBM 7094 digital computer to produce animated films. The coordinates of points are recognized by the computer and used to position the electron beam of a CRT. A picture is traced on the face of the tube. A 16-mm camera, also under the control of the computer, photographs the surface of the tube while the picture is being traced. The coordinates determined by the computer are automatically connected to produce the desired film. Applications are in the areas of satellite stability and orbit presentation, physical motion of objects, and display of the development of three-dimensional objects.[3]

(c) Rand Corporation has developed the RAND Tablet, which has been operational at Rand since September 1963 and has recently been installed for research purposes at several institutions. An electric pencil is used to write on a ten-inch by ten-inch conductive sectored tablet. The tablet is connected to a computer and to an oscilloscope display which provide a composite of the current pen position and meaningful track history. The Rand Corporation is currently developing a method by which characters can be recognized as they are drawn on the tablet, as well as on the oscilloscope display unit.[4]

(d) Stromberg-Carlson Corporation is continuing to develop its microfilm printer-plotters for additional engineering applications. More than forty operational systems are translating computer-generated data into drawings, annotated graphs, etc. The system converts binary-coded input signals into images on the face of a shaped-beam tube.[5]

(e) MIT's Lincoln Laboratory has developed SKETCHPAD. SKETCHPAD deals primarily with geometry. Sketches are drawn on a CRT with a light-sensitive pen or an electronic stylus. Freehand irregularities are automatically transformed into neat lines. SKETCHPAD can be employed to animate a drawing or to rotate parts of a drawing at high speeds or in slow motion so that linkages may be studied in action. Electric-circuit diagrams can be animated to show current flow. Other operations are available. SKETCHPAD has three-dimensional display characteristics. Front, plane, and side views as well as a rotatable perspective view of an object can be presented.[6] Roberts of MIT generates views of solid three-dimensional objects with all hidden lines removed.[7]

(f) General Motors and IBM have cooperated in the development of the Design Augmented by Computer System (DAC-1). An electrical probe contacts a resistive X-Y coding on the front of a Scan CRT, which has been developed by IBM. The X-Y voltage pulses are transformed by the computer into the coordinates of the spot under the probe. After the computer determines the coordinates, an X appears on the screen under the probe. The operator may write or draw on the CRT. Three CRT's are involved: a Scan CRT for automatically converting drawings into digital coordinate information, the console ten-inch CRT with the resistive probe input for drawing, and a CRT from which the display is photographed and recorded on microfilm.

The Scan CRT for digitizing drawing information has a dynamic threshold adjustment which is set by the computer as each line is being scanned and digitized. The output of the DAC-1 CRT at General Motors is ten inches in diameter and has a resolution of 1:2000 obtained by a daily calibration, correction and adjustment procedure. The DAC-1 output CRT has a fourteen-bit drive. After calibration and correction, the accuracy is eleven bits. DAC-1 personnel have developed a drawing control system. General Motors uses the system in designing various parts of automobiles.

(g) The Computer-Aided Mechanical Engineering Design System (COMMEND) is a problem-oriented system and works equally well in the design of a card punch, a printer, a document transport, or similar machines. The primary objective of COMMEND is to assist design engineers with high speed computations and to provide large memory capacities by means of a computer. The program determines a mathematical model of the requirement and performs tolerance, dynamic, and wear analyses. The computer determines a sensitivity coefficient for each dimension and tolerance so that critical situations are recognized by the engineer. The system reduces human error in design calculations and data handling. All

analyses are documented and stored for retrieval purposes.[8]

(h) Douglas Aircraft has a graphic research program to assist in design, documentation, manufacture, and test of electronic hardware. A major research goal is to solve wiring interconnection problems. A three-level wiring program has been developed. The first level performs chassis wiring from point to point and specifies size and length of wire, number of wire wraps that can be mounted on a given terminal, and associated documentation. The second level performs harness wiring (branched wiring assembly), and the third level designs cables. In the cable program, the computer first tries to satisfy current requirements by using a previously designed cable or its closest matching subset. Douglas uses an SC 4020 to show wire build-ups on wire-wrapped pins. Personnel monitor this operation. Plans for expansion involve generalization of programs and program modules to allow for more efficient programming (less reprogramming for new requirements) and to facilitate more efficient program organization (master control program module sequence, per run).[9]

(i) The Norden Division of United Aircraft Corporation has developed a computer-aided program to design integrated circuits. Much of their effort has been directed toward improved methods for synthesis, analysis, layout, and packaging of electronic circuits. Circuit design and connectivity are inputs to a computer, which performs a pattern and diffusion analysis of the data. The size and shape of the mask for each part as well as routing and wiring information is determined. This user-oriented system describes the circuit node by node. Engineers determine values and tolerances. The Norden program utilizes a designer's initial thoughts about parts layout but completes the layout automatically. The system currently uses a computer-controlled plotter for graphical display and for automatic preparation of mask artwork. Plans for future developments include on-line operation with graphical display and manipulation of circuit layout patterns on a CRT display with a light pen.[10]

(j) MIT personnel have established projects MAC (Multiple Access Computer) and AED (Automated Engineering Design), and offer their automated design tapes to the general public. MIT is currently developing AED Graphics. This graphic system uses an Algorithmic Theory of Language which requires no distinction between verbal language communicated through a typewriter keyboard and graphical language communicated by means of push-buttons and a light pen. This graphic system is called Computer-Aided Design Experimental Translator (CADET). CADET permits the user to apply verbal or graphical languages of his choosing.[11] Project personnel extend an invitation to industries: Send a representative to Boston to study the automated design project and participate in its design and development. The cost to a company is the salary and travel expense of its representative. In return, the company acquires the latest design automation information available at MIT.

Computer-controlled drafting machines can be obtained from Gerber Scientific Instruments Company, CalComp, Universal Drafting Machine Company, Benson-Lehner, EAI, IBM, G. Coradi Ltd., Dennert and Paper (Aristo-Werke), Sheffield (Ekstrom-Carlson), Visual Inspection Products, and others. The machines can be used for large-scale drawings. The time required to produce a drawing by means of a computer-controlled drafting machine normally ranges from thirty minutes to ninety minutes.

Visual display systems can be obtained from Stromberg-Carlson, Straya Industries, Benson-Lehner, Data Display, IBM, ITT, GE, RCA, Bunker-Ramo, Philco-Ford, Sanders Associates, Raytheon, and others.

It has been established that displays and graphic input-output devices, virtually undeveloped as recently as 1963, will constitute approximately eleven percent of the information processing equipment cost for a typical manufacturing company by 1973.[12]

Computer graphics usually involve a CRT, a light pen for drawing and manipulation of the image, an associated console, and switches for control of frequently-used subroutines and macro instructions. This peripheral equipment is linked to a computer. Communication through graphics is inherently structural; it is reasonable that a truly fundamental effort toward implementation of man-computer conversation is the most important step taken in computer technology.[13]

Graphic systems are particularly suited to design tasks because of their capability to shorten cycle time and facilitate change. Graphic data processing can rapidly provide accurate designs. Graphic systems can translate graphic and alphanumeric information and can process words, tables, and equations as well as graphs, drawings, and charts. Until recently, these

tasks appeared to be beyond the capabilities of a computer. Computer graphics will continue to expand. Handling of information graphically in computer systems will be as normal in a few years as handling of characters is today.[14]

Experimental procedure

Background

The major problems encountered during attempts to improve the accuracy of the SC 4020 visual display system were:

1. Development of a procedure to accurately measure errors.
2. Identification of a stable film base that did not change dimensions as variations occurred in the environment.
3. Determination and isolation of causes of deflections.
4. Determination of mathematical functions to correct the CRT display.
5. Dynamic focus.
6. Tangential errors.
7. Interaction errors.

The major causes of problems encountered during attempts to improve the accuracy of a visual display system were:

1. Astigmatism.
2. Pincushion correction.
3. Effect of the earth's magnetic field.
4. Exponential stretch of the film.
5. Quanta noise (error due to the size of the raster).

Preliminary investigations

Preliminary investigations led to the supposition that if errors in the SC 4020 system could be measured accurately, then mathematical techniques could be developed to predict the locations of errors on the CRT screen. Predictable errors could be corrected. Discovery that a microfilm image could be enlarged to exact dimensions led to recognition that designing, programming, and microfilming a standard matrix test pattern should be possible. The resulting image could be enlarged and the test pattern could be measured.

Development of a standard calibration pattern

A standard matrix test pattern which covered the CRT screen was designed and seemed to be the most valuable technique to use in checking for errors in orthogonality (skewness) and in checking the film transport mechanism.

Collection of data

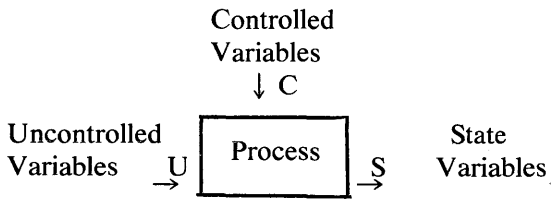
The appendix contains pictures of the equipment used in this research. Data for the study was collected by the following procedure:

1. The magnetic tape labeled "4020 Calibration Pattern" was processed at least twice daily on the SC 4020 machine.
2. Test patterns were run immediately before and after the regularly scheduled SC 4020 operations.
3. The test pattern was run both before and after any adjustments were made to the SC 4020.
4. Thirty-five millimeter film was used.
5. A note indicating the date and time of the test was attached to the exposed film. The film was developed by a microfilm laboratory.
6. The microfilm image of the test pattern was enlarged and the coordinates of each symbol were measured by a Ferranti machine.

Mathematical analysis and programs

Producing an accurate image with a computer-controlled CRT was assumed in this study to be a process model building endeavor initiated to solve a process control problem. A process simulation analysis was performed.

Many processes can be represented by displays similar to the following diagram:



Process parameters which can be adjusted are classified as controlled variables. Process parameters which cannot be adjusted are regarded as uncontrolled variables. Variables which are the result of process action on the controlled and uncontrolled variables are referred to as state variables.

The controlled variables in the model developed for this research were the coordinates of points to be accurately positioned by the SC 4020 display system. The coordinates could be changed before they were entered in the digital to analog converter of the SC 4020. The uncontrolled variables were the parameters associated with the hardware of the system. At any given time, the uncontrolled variables which affected CRT performance could be measured.

The dynamic state of a process control model with n state variables can be characterized by a system of n differential equations. One equation is required for each state variable. This implies that n equations of the following form can be used.

$$\frac{ds_i}{dt} = f_i(s_1, \ldots s_n, c_1, \ldots c_m, u_1, \ldots u_p)$$

for i = 1, n, where m = the number of controlled variables and p = the number of uncontrolled variables.

The following notation can be used. Define S, C, and U as follows:

$$S = \{s_1, s_2, \ldots s_n\}$$
$$C = \{c_1, c_2, \ldots c_m\}$$
$$U = \{u_1, u_2, \ldots u_p\}$$

Then $\dot{S} = F(S, C, U)$ where the dot above S indicates differentiation with respect to time.

Changes in either controlled or uncontrolled variables can produce changes in state variables. Corrective action may be required. Such corrective action is called system control. In practice, only two control policies exist: feedback control and feedforward control. Perfect feedforward control requires a perfect process model and precise measurement of uncontrolled variables. Feedback control does not provide corrective action until after an error in a state variable has been observed. Since one aim of this research was to correct errors in the CRT electron beam before the beam was generated, a feedback control design could not be employed. A system was developed for feedforward control.

In this research, the state variables S were measured and compared to the theoretical or desired condition S_T. The error E defined by the expression $E = S_T - S$ was determined and routed to a statistical computer program. The output of this program was applied to the controlled variables C and significantly reduced the magnitude of E.

System inaccuracy due to uncontrolled variables was measured as described in the section of this paper entitled "Collection of Data." The measurements of inaccuracy were used in statistical programs to simulate positioning of the coordinates and predict system performance. The output predictions governed manipulation of the controlled variables C to produce the desired state variables S.

In summary, the model was a feedforward control system developed to prevent deviation of actual from theoretical in positioning performance of the CRT.

Computer process control models were derived from a multiple regression analysis of error measurements. Such statistical models provided an explicit description of the process and were modified when deviations in positioning occurred.

The model parameters were adjusted periodically as a result of comparisons of current model behavior to actual process behavior. A multivariate prediction computer program was used to make these comparisons.

Functional relationships between final and initial conditions of the process were determined. A regression analysis was performed on the output data. The analysis revealed independent variables which most significantly reduced the errors. Errors in the CRT display were revealed to be systematic in nature. Recognition that such systematic errors must be correlated throughout the calibration matrix led to the supposition that the errors could be predicted and subsequently corrected.

Multiple regression analysis

Flow charts, statistical methods, and computer programs used in this research are discussed in this section. Techniques of multiple regression are well known; no attempt is made to present a complete discussion of regression analysis. Some basic discussion of multiple regression is presented to clarify differences between regression techniques used in this research and conventional regression techniques.

Multiple regression analysis is used primarily to obtain the equation which best fits a given set of data. The equation is typically of the form:

$$y = b_1x_1 + b_2x_2 + b_3x_3 + \ldots b_nx_n + a. \quad (1)$$

y is commonly referred to as the dependent variable, the criterion variable, or the predicted variable. $\{x_i: i = 1,n\}$ is the set of variables of prediction or the set of independent variables. Classification of these variables as independent does not imply that the variables are uncorrelated. $\{b_i: i = 1,n\}$ is the set of coefficients of the regression equation. a is a constant.

The solution of a particular equation may be interpreted as the value of the coefficients for a specific data sample. The standard error of each coefficient is calculated. These calculations provide a measure of the reliability of the coefficients and are the bases for inferences regarding the parameters of the population from which the data sample was obtained.

Multiple regression procedures were employed in this project but were adapted to fit nonlinear equations of the form:

$$y = b_1z_1 + b_2z^2{}_1 + b_3z_1z_2 + \ldots + b_nf_n(z_1,z_2,\ldots z_n) + \quad (2)$$

Equation (2) can be obtained from equation (1) by a process of transgeneration where

$$x_1 = z_1$$
$$x_2 = z^2{}_1$$
$$x_3 = z_1z_2$$
$$\vdots$$
$$x_n = f_n(z_1, z_2, \ldots z_n).$$

A conventional stepwise multiple regression analysis computer program was modified for use in this research project. The original program was designed by

M. A. Efroymson.[15] Multiple linear regression equations were derived by the computer in a stepwise manner as follows:

$$Y_1 = a+b_1x_1$$
$$Y_2 = a'+b'_1x_1+b_2x_2$$
$$Y_3 = a'' + b''_1x_1 + b'_2x_2 + b_3x_3$$

If the addition of x_j contributed significantly to error reduction, this variable was added to the regression equation. New coefficients were calculated whenever a variable was added. In general, each succeeding equation contained a new variable. The variable which achieved the greatest improvement in the fit of the equation to the data sample was selected. This variable produced the best estimate of the criterion variable as evidenced by the sum of the squares of the errors. It possessed the highest partial correlation to the dependent variable partialed on the variables which had previously entered the regression equation. It was also the variable which had the highest F value when added to the equation.

The output of the computer program showed:

1. Sum of each variable
2. Sum of squares and cross products
3. Residual sums of squares and cross products
4. Means and standard deviations
5. Intercorrelation matrix

At each step, the output showed:

1. Multiple R
2. Standard error of the estimate
3. Constant term
4. For each variable in the regression
   (a) Regression coefficient
   (b) Standard error of the coefficient
   (c) F to remove
5. For each variable not in the regression
   (a) Partial correlation coefficient
   (b) Tolerance
   (c) F to enter

In the stepwise procedure, the variables were divided into disjoint sets:

$$X_1 = \{x_{11}, \ldots x_{1q}\}$$
$$X_2 = \{x_{21}, \ldots x_{2r}\}$$

where X_1 was the set of independent variables in the regression equation and X_2 was the set of independent variables not in the regression equation and all dependent variables. If a variable was no longer significant, that variable was removed before another variable was added. Only significant variables appeared in the final regression equation.

The symbols used in the subsequent discussion should be interpreted as follows:

N = the number of cases of data or the number of observations

m = the number of variables (input and transgenerated variables and the dependent variable)

x_{ip} = the pth value of the ith variable

$x_{mp} = y_p$ = the pth value of the dependent variable

$\bar{x}_i$ = the mean of the ith variable

s_{ij} = the residual sum of squares and cross products of the ith and jth variables, i.e., $s_{ij} = \sum_{p=1}^{N} (x_{ip}-\bar{x}_i)(x_{jp}-\bar{x}_j)$

$\sigma_{ii} = \sqrt{s_{ii}}$ (This is *not* the standard deviation.)

r_{ij} = the correlation coefficient between the ith and jth variables. The ordered array of the m^2 coefficients forms the simple correlation matrix r.

c_{ij} = the element in the ith row and jth column of the inverse matrix of the matrix r

N_{min}, N_{max} = selected independent variables

V_{min} = the increase in variance obtained by deletion of the variable N_{min} from the regression equation

V_{max} = the decrease in variance obtained by addition of the variable N_{max} to the regression equation

b_i = the estimated value of the coefficient of the ith variable

s_y = the standard deviation of the dependent variable

s_{b_i} = the standard error of the coefficient of the ith variable

y_p = the predicted value of the dependent variable for the pth observation

D_p = the discrepancy between actual and predicted values of the dependent variable on the pth observation, i.e., $D_p = y_p - \hat{y}_p$

F_1 = the F value established as the criterion for entrance of a variable in the regression equation (for this project, $F_1 \geqq 2.5$)

F_2 = the F value established as the criterion for deletion of a variable from the regression equation (for this project, $F_2 \leqq 2.5$)

$$E_p = y_p - \bar{y} - \sum_{i=1}^{m-1} b_i (x_{ip} - \bar{x}_i) \text{ for } p = 1, N$$

$$E^2 = \sum_{p=1}^{N} \left\{(y_p - \bar{y}) - \sum_{i=1}^{m-1} b_i (x_{ip} - \bar{x}_i)\right\}^2$$

To determine b_1, $i = 1, m-1$, which minimize E^2, E^2 is differentiated partially with respect to each b_i. Normal equations are then obtained by equating each partial derivative to zero. The normal equations can be written in the following concise form:

$$\sum_{j=1}^{m-1} \left\{ \sum_{p-1}^{N} (x_{ip} - \bar{x}_i)(x_{jp} - \bar{x}_j) \right\} b_j =$$

$$\sum_{p-1}^{N} (x_{ip} - \bar{x}_i)(y_p - \bar{y})$$

The set of m-1 simultaneous linear equations in b_i can be solved by several methods. In this research, linear transformations were applied to the partitioned matrix illustrated as follows:

$$M = \begin{vmatrix} S & H' & I \\ H & K & D \\ -I & B & C \end{vmatrix}$$

B was a one-column matrix. D was a one-row matrix. K was a constant illustrated as follows:

$K = \Sigma (y_p - \bar{y})^2 = \Sigma (x_{mp} - \bar{x}_m)^2$ where $x_{mp} = y_p$. I was the identity matrix. H was a one-row matrix illustrated as follows:

$(H)_{1j} = h_{1j} = (x_{jp} - \bar{x}_j)(y_p - \bar{y})$ for $j = 1, m-1$. $h'_{j1} = h_{1j}$ defined the elements of the matrix H'. The element s_{ij} of S was the deviation sum of squares and cross products as defined previously.

Regression coefficients were entered into the B matrix. The D matrix was obtained by first transposing the B matrix and then multiplying the new matrix by -1. That is, $d_{i1} = -b_{1i}$.

The juxtaposition of the submatrices S, H, H', and K formed an m by m matrix and could be partitioned in any manner. This m by m matrix was normalized and became an intercorrelation matrix.

As the calculation proceeded, the S matrix was inverted. The D matrix replaced the H matrix. The B matrix replaced the H' matrix, and the C matrix replaced the S matrix. The C matrix contained the inverse of the portion of the S matrix which corresponded to the variables in regression at that time.

Correlation coefficients needed at various steps in the computation were extracted from the M matrix. As the addition of variables proceeds, the elements of S become partial correlation coefficients; because all included variables being held constant.

The computer initiated the regression procedure by examining the intercorrelation matrix and selecting the independent variable which had the highest correlation with the dependent variable. An F test was made to determine whether the contribution of the selected variable to the prediction of the dependent variable would be significant. The determination was made by examining the reduction in the standard error of the dependent variable. If the reduction would be significant, i.e., above an established F level, then the program retained this variable. The major steps performed by the computer are described in detail within the discussion of system flow charts. Essentially,* the computer program proceeded as follows:

1. The transformations were generated.
2. The sum of each variable was accumulated.
3. The raw sum of squares and cross products were calculated.
4. Deviation sums of squares and cross products were calculated. Each element of the intercorrelation matrix was calculated.
5. Means and standard deviations were determined.
6. The simple correlation coefficients were calculated by evaluating the following equation:

$$r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}\, s_{jj}}}$$

As an alternate procedure, a raw score formula could have been used.

7. The estimated population value of the standard deviation of y was obtained as follows:

$$\tilde{\sigma}_y = \mathcal{S}_y \sqrt{\frac{N}{N-1}}$$

8. The estimated population value of the standard error of the estimate for y was determined as follows:

$$\tilde{\sigma}_{est.y} = \mathcal{S}_y \sqrt{1 - R^2} \sqrt{\frac{N}{N-1-q}}$$

where q was the number of variables in the regression equation at the time of the calculation.

9. The coefficient b_i was determined by evaluating the following expression:

$$b_i = b_{im} \frac{\sigma_m}{\sigma_i}$$

10. The standard error of b_i was determined by evaluating the equation:

$$s_{bi} = \tilde{\sigma}_{est.y} \sqrt{c_{ii}}$$

The flow chart follows. At certain branch points in the chart a finite number of alternative statements

*Although the output conformed to the following steps, the program did not follow precisely all formulas shown.

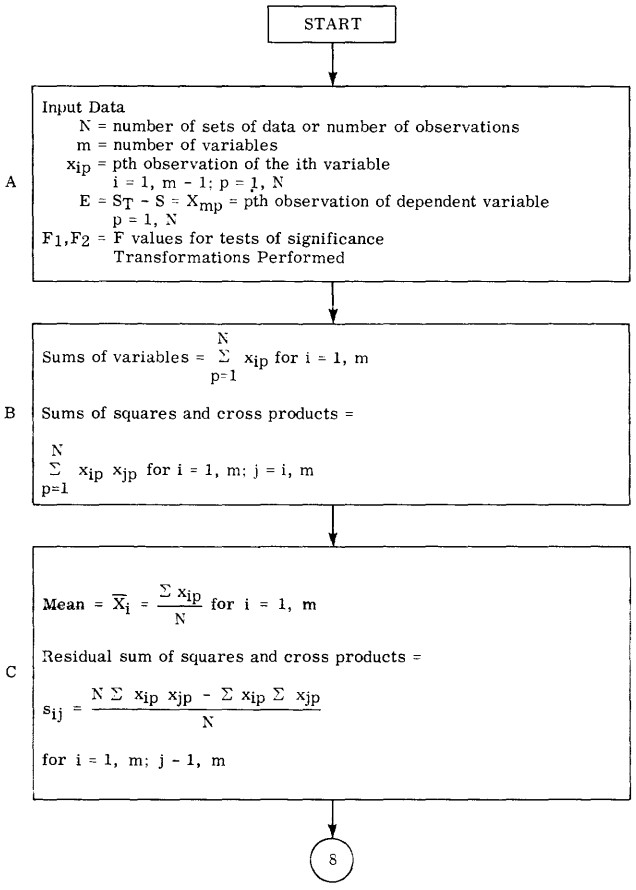
are specified as possible successors. One of these successors is chosen according to criteria determined in the statement preceding the branch point. These criteria are usually stated as a comparison of a specified relation between a specified pair of quantities. A branch is denoted by a set of arrows leading to each of the alternative successors, with each arrow labeled by the comparison condition under which the corresponding successor is chosen. The quantities compared are separated by a colon in the statement at the branch point. A labeled branch is followed if and only if the relation indicated by the label holds when substituted for the colon.

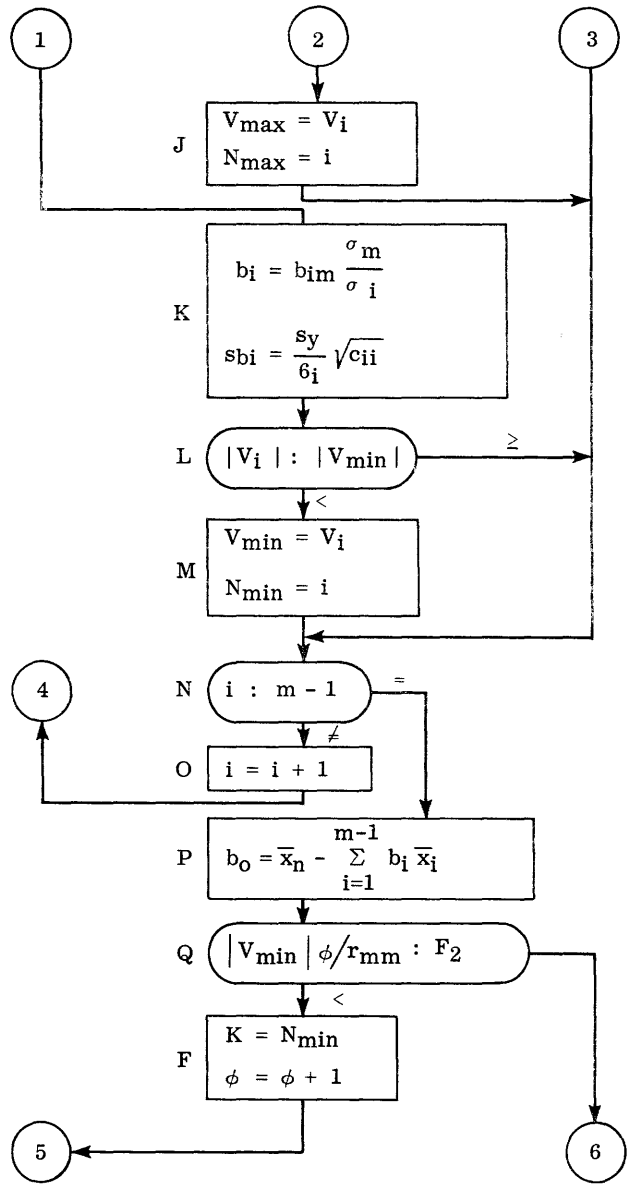
Block A, Block B, and *Block C:* The content is self-explanatory.

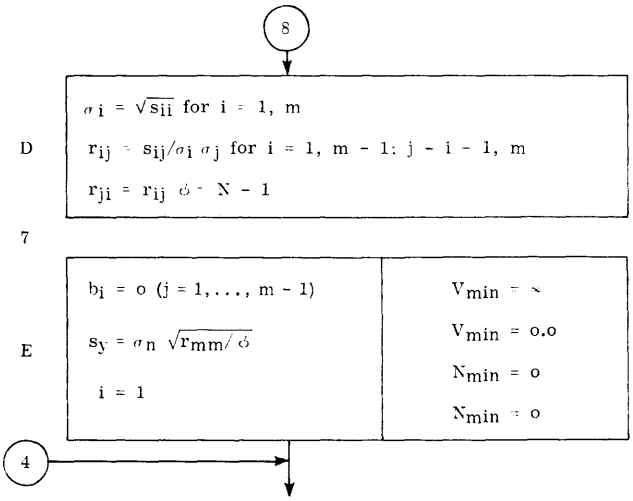
Block D: The simple intercorrelation matrix was obtained by normalization of the residual sums of squares and cross products.

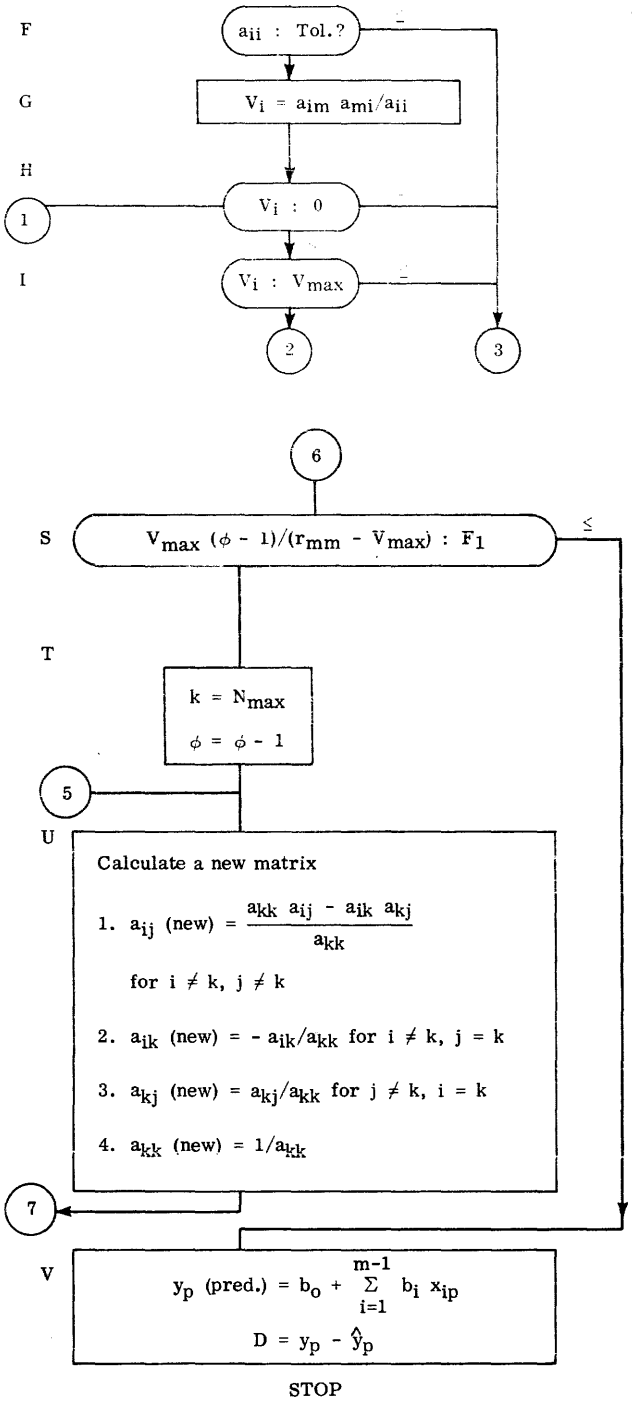
Block E: Procedures to select the variable to be added to the regression equation were initiated.

Block F: The size of a_{ii} was checked; this check reduced the probability that an independent variable which was a linear combination of a variable more highly correlated with the criterion variable would enter the regression equation. A value of .001 was assigned to Tol.









Block G: The quantity V_i was calculated.

Block H: The sign of V_i was determined. V_i positive implied that X_i was not in the regression; V_i negative implied that X_i was in the regression at the time of the test.

Block I and *Block J:* The variable X_i which would cause the greatest reduction in the variance was selected.

Block K: The standard error and the regression coefficient were calculated for each variable currently in regression.

Block L and *Block M:* The X_i which contributed the least toward reduction of variance was identified.

Block N, Block O, and *Block P:* The content is self-explanatory.

Block Q: The variance contribution of each variable was measured; insignificant variables were deleted from the regression.

Block R: The content is self-explanatory.

Block S: A test determined whether the variance reduction obtained by adding X_i to the regression would be significant.

Block T: If X_i would contribute significantly, the variable was entered into the regression equation. The number of degrees of freedom was reduced by one.

Block U: A new matrix was calculated and replaced the original matrix in the computation. The elements were normally entered into the I matrix. I was generated except for the elements in the row and column corresponding to the subscript of the variable added to the regression. The new matrix was computed by the following procedure:

1. $a_{ij} = r_{ij}$ when both x_i and x_j were not in regression.
2. $a_{ij} = b_{ij}$ when x_j was not in regression and x_i was in regression.
3. $a_{ij} = d_{ij}$ when x_i was not in regression and x_j was in regression.
4. $a_{ij} = c_{ij}$ when both x_i and x_j were in regression.

The C matrix was the inverse of the r matrix for all i's and j's in regression.

Block V: The predicted values of the dependent variable were calculated. The computer determined the actual value of the dependent variable, the predicted value of the dependent variable, and the difference between the two values.

Cross validation

The results of the multiple regression program show how the regression equations performed for the data sample from which they were derived. The way such equations perform for an independent sample is more important. Major concerns of this research included not only how well regression equations predicted for the particular sample from which they were derived, but also how well and for how long the equations provided reliable predictions related to new sets of data.

The correlation between actual and predicted errors serves as an index of the accuracy of the prediction determined by the equation derived from the same data sample. This correlation cannot be used to assess the accuracy with which this same equation can be applied to a different data sample. To assess the accuracy with which a regression equation derived

from one set of error measurements can be applied to another set of error measurements not satisfying the conditions of the regression model, it was necessary to use an index based on actual errors of etimates (i.e., $y - y$).[16]

Cross validation procedures for this research were contained in a multivariate prediction computer program. The final regression equation was applied to a new sample of error measurements. Values of the dependent variable were predicted by the computer. The accuracy of these predictions was evaluated by computing the product moment correlation coefficient (Pearson r) between the predicted and actual values.

The regression equations used in the multivariate prediction program were derived from data collected daily throughout one week. These equations were applied to predict the errors of a new data sample.

The results of analysis

This research project provided the bases for responses to questions proposed earlier in this paper. Several questions which arose during the research project were successfully answered.

The aim of this research was to demonstrate that mathematical techniques can be applied to increase the accuracy of an electronic device despite hardware limitations. This goal was achieved. Such techniques were applied to the SC 4020. Accumulated results show that the SC 4020 can be used to produce accurate engineering artwork, i.e., that an *inaccurate* computer peripheral device can provide *accurate* data output. This affirmation implies that a Cathode Ray Tube (CRT) connected to a computer can function as an accurate high speed artwork generator.

Results of this research indicate that artwork generation by a visual display system is possible. Location accuracy of $\pm .004''$ for $8'' \times 8''$ artwork and $\pm .0025''$ for $5'' \times 5''$ artwork can be obtained.

Equations were derived to predict the locations and magnitudes of errors in the output of the SC 4020 system. Errors transmitted by the CRT were predicted. A computer supplied modified, seemingly inaccurate data to a peripheral device. Accurate data output resulted. Errors of linearity were predictable in both X and Y directions. Exponential terms in the prediction equation compensated for film stretch due to irregular transport.

Initially, the data obtained from the morning test was used to predict afternoon performance of the tube. A cross-validation computer program was used on the data from the afternoon test. After the proper generating functions had been derived, the correlation coefficient on the cross-validation program was as high as .973. Calibration equations derived from data obtained during a week of testing were used to predict tube performance eight days later. The correlation coefficient obtained was .968; the equations which were originally obtained for corrections were valid eight days later. Tube maintenance was performed during this period. Thus, the tube appeared stable during the elapsed time period. Recalibrations of the tube less frequently than once a week would perhaps suffice.

The range of errors before correction was typically .025 inches to $-.080$ inches. The range of errors after correction was $\pm .004$ inches. The maximum error before correction was .087 inches. The maximum error after simulation correction was .004 inches. The average degree of error before correction was .0150 inches. The average degree of error after simulation correction was .0015 inches.

The correction equations derived by the computer program were applied to data obtained from the Ferranti operation. All symbol coordinates were recalculated and compared to theoretical coordinates. Computer programs determined where the symbols would have been located by the SC 4020 if correction equations had been applied to original coordinate information. The error measured by the Ferranti machine, the calibration correction for each position, and the difference between corrected and theoretical coordinates were determined by the computer. An example of the error distribution before and after correction is displayed on the accompanying graph.

The simulation correction procedures were used primarily to derive the equations necessary for correction of actual data. The calibration equations were employed to generate accurate circuit card artwork. The SC 4020 software was modified to include compensated deflection voltages. The deflection voltages were corrected by adding the calibrated errors to the corresponding given coordinate pair.

That is, $(X, Y)_{\text{deflection}} = (X, Y)_{\text{given}} + E$

or, $(X, Y)_{\text{calibrated}} = (X, Y)_{\text{original}} + E$

or, the plotted (X, Y) = the theoretical (X, Y)

or, the adjusted coordinate positions = the result of application of the derived equations to original coordinates.

The calibration equations were applied to given coordinate points: a predetermined correction was added to the deflection voltages for each point to be located

by the SC 4020. The given coordinates were converted to calibrated deflection voltages. The result was accurate computer graphic output.

SC 4020 production of engineering drawings would be approximately 500 times faster than existing computer-controlled drafting machines. More important than speed, however, is the possible on-line production of circuit cards by application of the techniques developed by this research.

Honeywell is considering an advanced design automation program for on-line circuit card design/production in which a visual display system functions as a production tool. In this program the circuit card (without components) could be generated directly during the manufacturing process. Such a system would provide unlimited flexibility in producing circuit cards. One or several of each kind of card could be easily, quickly, and economically produced. The design engineer could retrieve a design from storage or could enter the coordinates of the card elements and parameters such as connections, into the computer, and the exposed circuitry would be automatically produced on a card. The research completed to date indicates that such on-line operation is possible. Each design produced would be stored for on-line retrieval. When the program outlined here is fully implemented, no draftsman will be required for circuit card design/artwork!

INTERPRETATIONS AND CONCLUSIONS

Research efforts portending significant improvements or alterations to existing conditions or acceptance of new techniques frequently introduce new questions which are as significant as answers provided for original queries. From this research evolve questions such as "In what other areas and by what procedures can the techniques of this research be applied to solve similar problems?"

The production of engineering artwork is only one example of how the results of this research can be applied to advance the electronic and computer industries. The initial goal in this application is to proceed directly from a computer to the artwork by means of an accurate CRT image.

Concepts analyzed and utilized throughout this study may contribute to process control or process model building applications in the following areas:

1. Air traffic control (collision avoidance).
2. Low approach aircraft landings.
3. Microwave video transmissions.
4. Off-course aircraft computers.
5. Graphic data concentration and transmission systems.

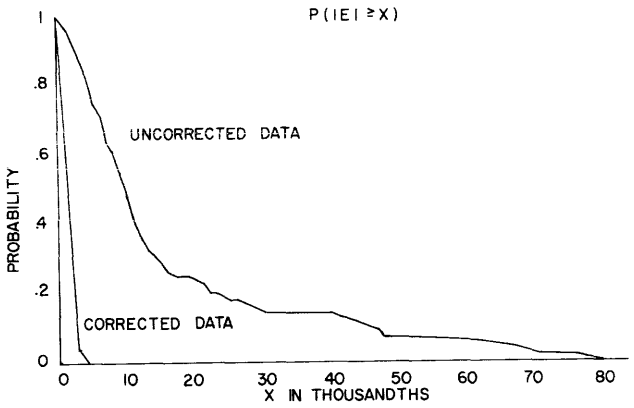


Figure 1 – Sample of error distribution before and after corrections

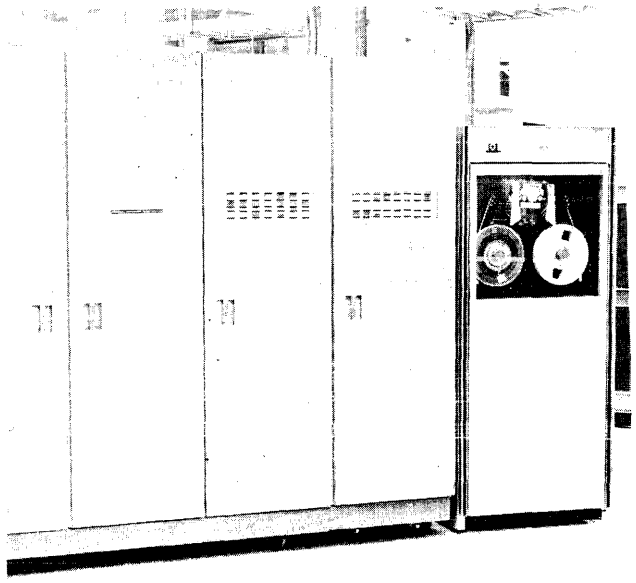


Figure 2 – Stromberg Carlson 4020 microfilm machine

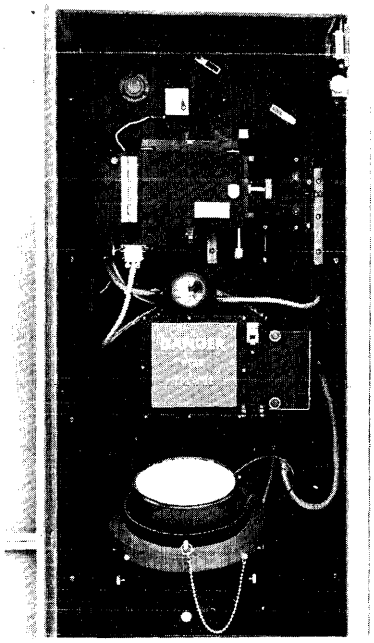


Figure 3 – Charactron tube and camera

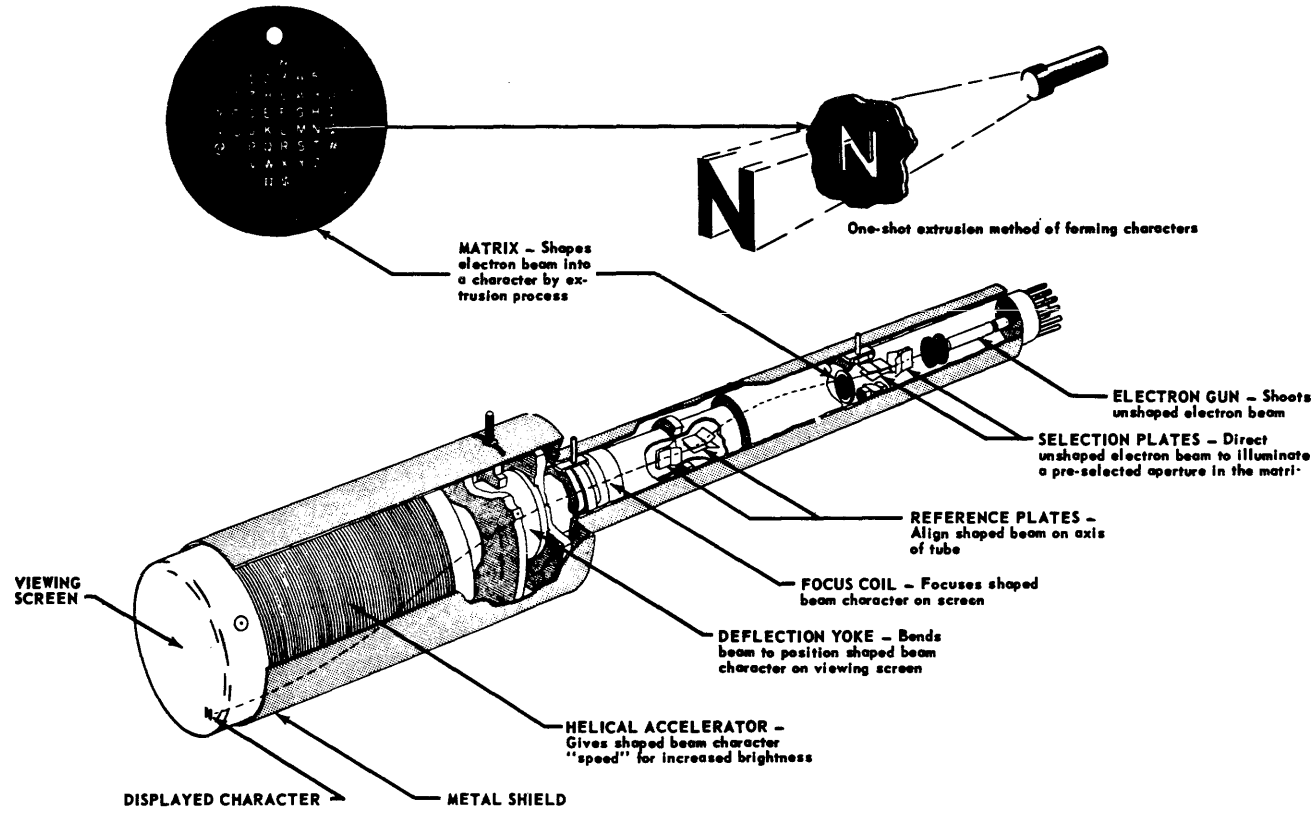


Figure 4 – Charactron shaped beam tube

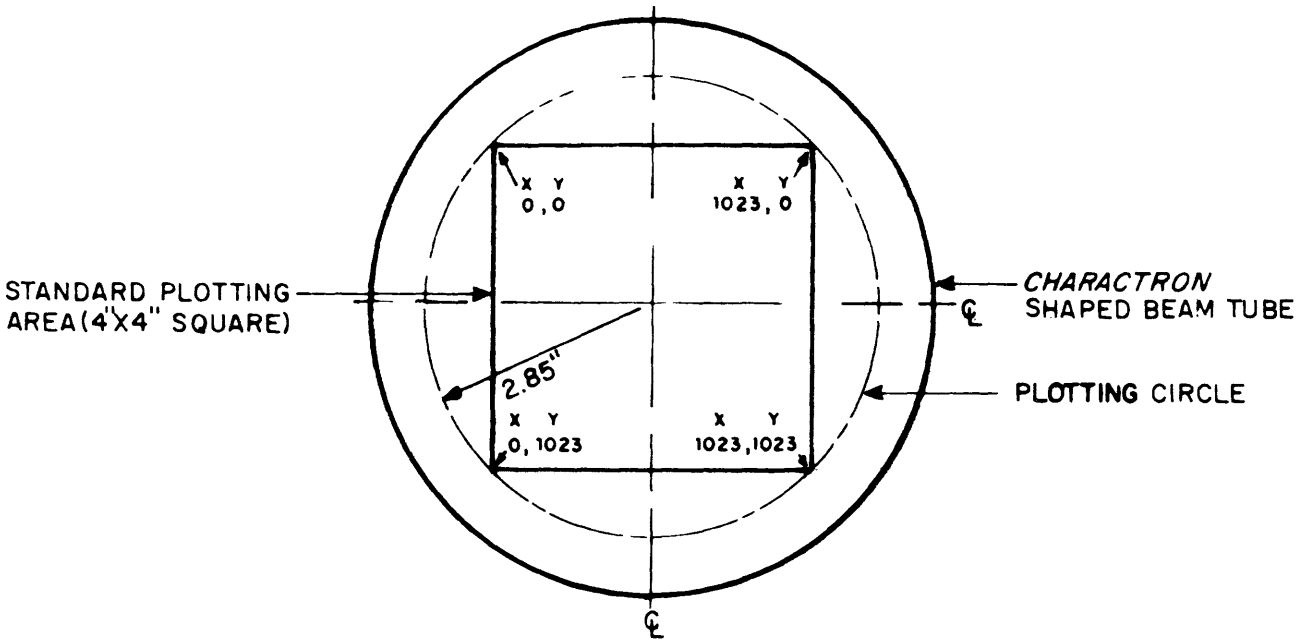


Figure 5 – Standard plotting format

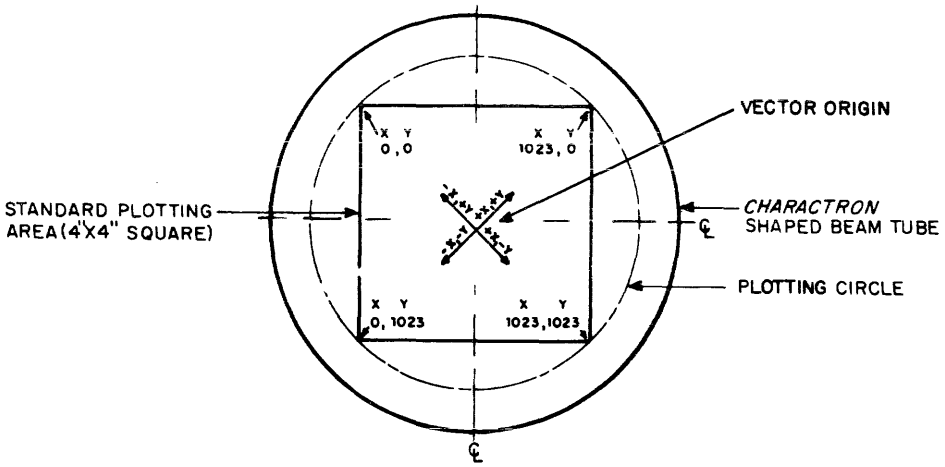


Figure 6—Vector component direction

Conjectures concerning how the techniques of this research can be applied are difficult to formulate. Clearly, an apparatus under consideration must be computer controlled and some aspect of its performance must be measurable.

The results of this research demonstrate that the accuracy of a visual display system can be improved despite hardware limitations. Potential benefits of a computer-controlled graphic device are comparable to benefits derived from the introduction and application of automatic data processing. The value of graphical information processing is difficult to access; only with the passing of time will this value be properly realized.

REFERENCES

1 C J FISH D D ISETT
Automated circuit card etching layout
Technical Memorandum Sandia Laboratory Albuquerque October 1965

2 Ibid 22

3 A MICHAEL NOLL
Computer generated three-dimensional movies
Computers and Automation 20 23 November 1965

4 M R DAVIS T O ELLIS
The RAND tablet A man machine graphical communication device
Instruments and Control Systems XXXVIII 101 103 December 1965

5 N WADDINGTON
Some applications of graphic data output
Computers and Automation 24 27 November 1965

6 I E SUTHERLAND
SKETCHPAD A man machine graphical communication system
AFIPS Conference Proceedings XXIII 329 346 Spring 1963

7 LAWRENCE G ROBERTS
Machine perception of three-dimensional solids
MIT Lincoln Laboratory Technical Report No 315 Boston Massachusetts Massachusetts Institute of Technology 1963

8 L F KNAPPE
A computer-oriented mechanical design system
Paper read at the Share Design Automation Workshop Atlantic City New Jersey 23 25 June 1965

9 D K FRAYNE
Three levels of the wiring interconnection problem
Paper read at the Share Design Automation Workshop Atlantic City New Jersey 23 25 June 1965

10 A SPITALNY W MANN G HARTSTEIN
Computer aided design of integrated circuits
Paper read at the Share Design Automation Workshop Atlantic City New Jersey 23 25 June 1965

11 D T ROSS
Current status of AED
Paper read at the Share Design Automation Workshop Atlantic City New Jersey 23 25 June 1965

12 JOHN DIEBOLD
What's ahead in information technology
Harvard Business Review XLIII No 5 77 September October 1965

13 S A COONS
Computer Graphics and Innovative Engineering Design
Datamation XXII No 5 32 34 May 1966

14 CHRISTOPHER F SMITH
Graphic systems for computers
Computers and Automation 14 16 November 1965

15 M A EFROYMSON
Multiple regression analysis in
Mathematical methods for digital computers
A Ralston and H S Wilf eds pp 191 203 New York Wiley 1960

16 P BLOMMERS E F LINDQUIST
Elementary statistical methods
Houghton Mifflin Company Boston 460 1960

A new printing principle

by W. H. PUTERBAUGH
National Cash Register Co.
Dayton, Ohio

and

S. P. EMMONS
Texas Instruments, Inc.
Dallas, Texas

INTRODUCTION

Previous printing applications have been largely confined to the processor printout, where high speed and multiple copies were necessary. These printers were largely mechanical impact devices operating near the edge of the performance expected of high-speed mechanisms. The few non-impact printers developed for these applications have not found general use because of unsuitable cost/performance ratios.

The recent development of time-shared systems has created a need for printers in the performance range of 20 to 50 characters per second for use in inquiry and terminal equipment. A recent survey of the peripheral market by Stanley Zimmerman showed that in 1955 this totaled $20 million or 19 percent of the computer market. In 1965, the peripheral market had increased to $1.68 billion or 51 percent. Mr. Zimmerman estimated that by 1975 peripheral equipment will have a total value of almost $4.5 billion or 69 percent of the computer market.

Three aspects of this need are reviewed. The first discusses the system trends towards electronic compatible printers as on-line multiple computer systems come into greater use. Problems associated with printers, such as speed and power requirements and reliability deficiencies, are discussed showing that a new printing principle is necessary to obtain the cost/performance needs of the future. This principle, that of thermal printing, is discussed together with a solid-state embodiment.

System trends

Definite trends in computer systems and especially in data processing systems indicate that time sharing of the central processor and its files is well under way. These types of systems require terminals and/or inquiry stations of many varieties to satisfy the need of the customer who is time sharing the processor. Figure 1 shows a time-shared system where many remotely located terminal units are connected by communication lines to a processor and its files. These terminal devices must have several capabilities, the minimum being a keyboard, a display and a printer. As this trend continues the major cost of the system will be in the terminal area. It is in this area, therefore, where cost and performance will be very important. Satisfactory costs, however, are difficult to obtain.

One of the most critical functions in the terminal area is that of the printer. The printing of alphanumeric data from electronic signals has always been difficult from a cost/performance point of view and even more disappointing from the standpoint of reliability. As system trends continue, the day of the "household computer" is not too far removed, where on-line availability of a processor will be possible. The printer associated with this device and other terminals must possess at least an order of magnitude increase in reliability over present mechanical printers.

Monolithic silicon technology has contributed much to improvements in logic and memory. The developments now occurring in the use of complex arrays of monolithic silicon promise even greater reliability and cost reduction. It would be logical, therefore, to examine this technology for its suitability to printing applications. It certainly is not obvious what relationship exists between printing and integrated circuit technology. In fact, at first thought, this ap-

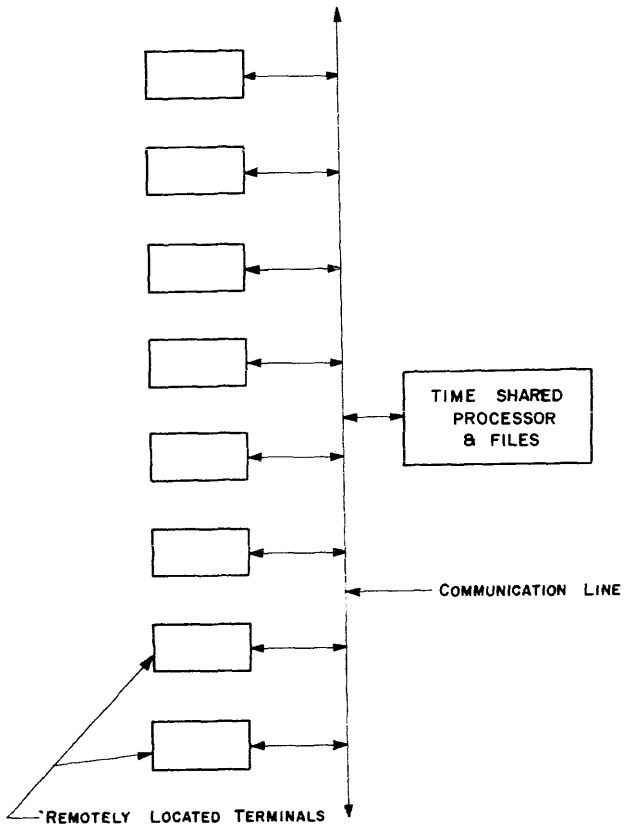


Figure 1 – Time shared system

pears to be the one function least compatible with the application of silicon technology.

The printer

The process of producing alphanumeric characters on paper from electronic logic signals must necessarily involve the change of electrical energy to a form of energy that will affect the paper in such a manner as to produce legible print. In the past, this has largely been done by controlling mechanisms whose power and speed differed markedly from the logic signals. Since these differences are the prime reason for the high cost of present printers, an economical method of transforming energy with better compatibility between the power and speed of the logic circuits and the printer is suggested. The most efficient and easiest energy transformation of electrical current is to that of heat. Papers have been available for some time which change from white to black when exposed to heat. A printing concept, then, involves selectively heating areas on some surface which is in contact with a thermally sensitive paper. Such a printer could be realized using the I^2R losses within resistors assembled to form a matrix of 5×7 or 5×5 heating elements. Selection of resistors within the matrix would produce the desired alphanumeric symbol.

Print head design

Normally, heat energy is not selected as the form of energy to be used when speed of any consequence is desired. Intuitively heat energy is thought to be inherently slow, but this is because the majority of everyday experiences involve sizable thermal masses.

Actually, the thermal time constant for a given structure depends upon its thermal mass, which includes the physical mass of the heated material, and the thermodynamic mechanisms acting to conduct energy away from it. A three-dimensional heat flow analysis is necessary to accurately predict the rise and fall times of temperature. For the purposes here, it is sufficient to generalize.

If thin-film resistors are to be used as heating elements, the thermal rise time of the element is largely dependent upon its physical mass. The choice of a substrate material upon which the heating element is placed largely governs the fall time of the temperature. The design of a printer, for a given speed of response, using the heat generated in a resistor, becomes one of selecting the most suitable combinations of resistor elements and substrates. Approaching the problem in this manner, without other considerations, resulted in the choice of thin-film resistors and ceramic (Al_2O_3) substrates. Each character, consisting of a 5×5 matrix of resistor elements, required 26 leads. A head assembly of five characters thus required 130 external interconnections. The complexity of these interconnections together with other considerations of importance, such as life, reliability and cost, have directed attention in an entirely different direction.

Reduction in interface complexity by silicon implementation

The integrated circuit art has demonstrated the economics of fabricating components within a monolithic structure of silicon. Such components as resistors, diodes and their interconnecting leads, have been routine for some time, and the feasibility of fabricating 125 resistors and diodes to produce a silicon structure containing five characters is attractive. Each character would be a 5×5 matrix of heater elements. A diode-resistor pair within each heater element would reduce the number of external leads for printing five characters sequentially from 130 to 30 as shown in Figures 2 and 3. Within the five character head, the characters would be printed sequentially. In a typical printing sequence the common for character #1 would first be activated and the appropriate drive lines corresponding to the correct alphanumeric symbol would be pulsed negatively. After a cool-off period, common #2 would be ac-

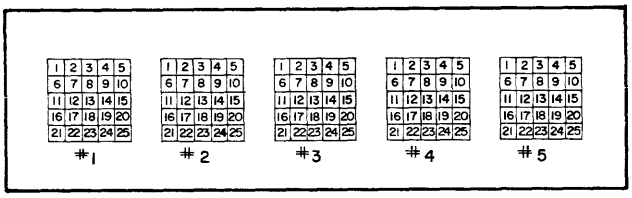


Figure 2—Character and element location as viewed looking at print

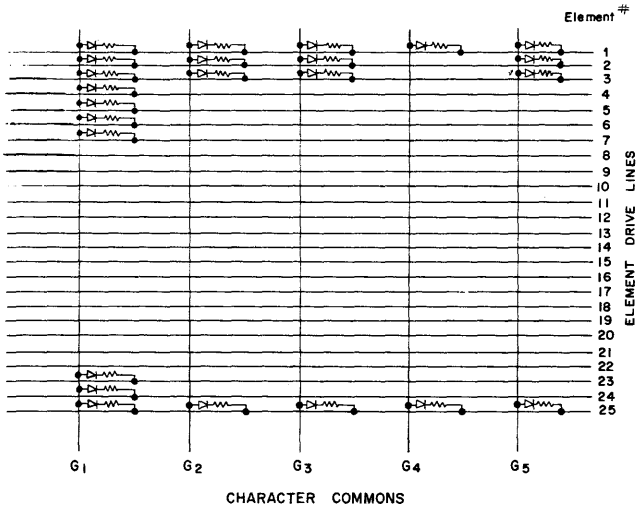


Figure 3—Interconnection scheme for 5×5 matrix printing head

tivated and a different combination of drive lines would be pulsed.

Various types of system drive schemes are conceivable. If the line in the machine is made up of an assembly of the five character heads, every fifth character could be printed in parallel. The highest speed printer, therefore, will be one in which a line will be printed in the time necessary to print five characters. In the lowest speed printer all characters in the line will be printed sequentially.

Print head fabrication

A prototype print head has been fabricated in monolithic form with the physical dimensions shown in Figure 4. Each heater element shown in Figure 3 is fabricated within the silicon mesas, shown in Figure 4, and each mesa contains a resistor-diode pair, produced by conventional integrated circuit diffusion techniques. The array was interconnected by normal evaporated leads. The leads are protected from paper wear because the silicon structure is mounted with the leads at the interface between the silicon and a ceramic substrate.

The structure of a silicon mesa mounted on a ceramic substrate was chosen to provide the desired thermal transient performance. Because silicon is a very good thermal conductor, the mesas are particularly necessary to provide thermal isolation between heater elements. As discussed earlier, the thermal characteristics of the interface and the ceramic as well as the thickness of the silicon mesa determine the temperature rise achieved for a given pulse width and power level.

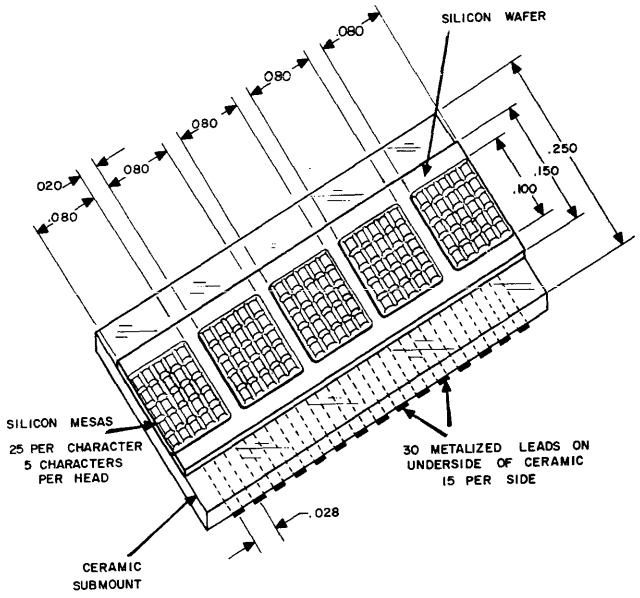


Figure 4—Construction of monolithic silicon print head

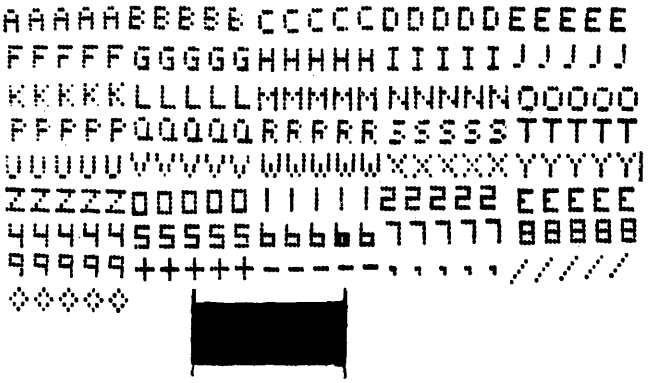


Figure 5—Example of print and printing head

Print head performance

Although the process is still in a development phase, much has been learned about the performance of such a head. The print head photographed in Figure 5 uses a 10-millisecond current pulse corresponding to a power level of approximately 700 mw/element to achieve a 200°C peak temperature. A 5-millisecond cool-off time, necessary to allow the diodes of the pulses character to recover before the next character is pulsed, results in a sequential character rate of 15 milliseconds per character. Typical prints are also shown in the photograph.

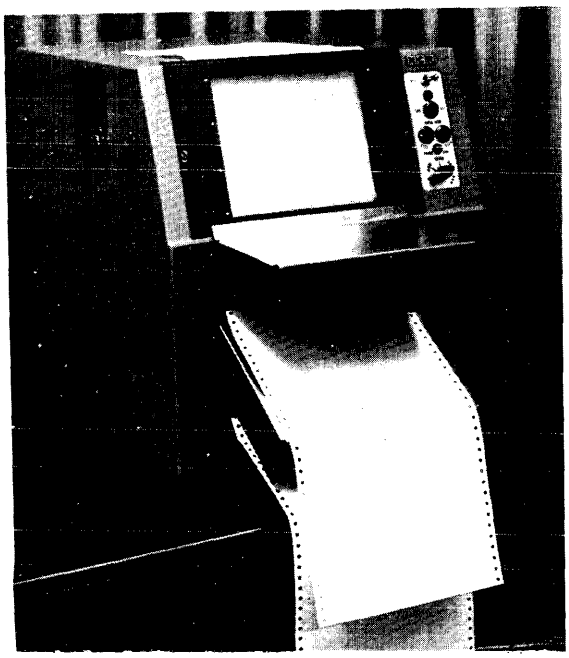


Figure 6—Thermal printer (80 characters per line—240 characters per minute)

Machine performance

No complete printers have yet been built using the silicon configuration. Many printers of various types have been built with other print head configurations using films of various types for the resistors. Figure 6 shows one such printer which operates in a serial-by-character mode, printing a line of 80 alphanumeric characters at a rate of 240 characters per second. The mean-time-between-failure of this printer is 2,000 hrs. at 30% duty cycle with a minimum life of the print head of 50×10^6 lines of print. Other printers have been delivered to operate in systems whose design objectives are in excess of 15,000 hrs. MTBF at 100% duty cycle. This illustrates the increase in life and reliability which can be expected of thermal printing.

The change of printing heads from films to monolithic silicon will give additional life and reliability. The silicon heads will provide much longer life for two reasons: (1) since the resistors and diodes are at the silicon-ceramic interface, as shown in Figure 3, mechanical wear of the silicon surface by the paper can proceed until practically all of the silicon is removed. The only difference noted during this wear process is an improvement in print intensity during the life of the print head; (2) the silicon is considerably thicker and harder than the films and provides more life in this manner. The monolithic silicon printing head will provide longer life and increasing print quality during its life without a change in the value of the resistors.

Most people observing the operation of the thermal printer wonder when the printer is going to start to operate without realizing that it is indeed already operating. The printing, of course, makes no noise in itself and paper spacing is the only mechanical motion necessary. Many techniques are available to reduce this noise and thermal printers have been built whose operation is only apparent by the gentle swoosh of the paper movement.

Thermal paper

The printer shown in Figure 6 produces an original and two copies. The use of thermal energy for printing assumes a paper sensitive to this energy with properties suitable for legal records. Several paper systems have been developed for various applications requiring up to twelve copies. These printing papers are difficult to tell from untreated papers by inspection or handling, and can be stored indefinitely at 65°C.

CONCLUSIONS

The feasibility of extending the electronic circuitry directly to the printed page by the solid-state embodiment described has been proven.

Tests have shown that reliabilities of the same order as experienced with integrated circuits can be expected.

Paper systems have been developed as an adjunct to this work which will provide copies, although in most terminal applications this will not be required.

Redesign of the printing head for production must be done before accurate cost estimates can be made. Life test data has not yet been obtained due to the length of time required for failure. The failure mechanisms are much the same as those in an integrated circuit having two levels of interconnections.

A time delay compensation technique for hybrid flight simulators

by V. W. INGALLS
The Boeing Company
Seattle, Washington

INTRODUCTION

A flight simulator or vehicle simulator of any type except those involving only a point mass, appears at first glance to be ideally suited for mechanization on a hybrid computer because of the difference in requirements of the rotational and translational equations of motion. Solutions to the rotational equations contain high frequencies, making the use of the analog computer best for solving these equations. The translational equations have mostly low frequency solutions but requires high accuracy so the digital computer is best. Also the digital computer is the best means of generating multi-variable functions such as aerodynamic coefficients. The equations to be solved and the conventional methods of mechanization of flight simulators are discussed by Fifer[1] and Howe.[2]

The division of the problem between the two halves of the hybrid computer is to solve the rotational equations mainly on the analog computer and the translational equations mainly on the digital computer. The rotational equations require multivariable functions so that part of the solution must be done digitally and because of the interaction of the translational equations with the rotational equations, part of the solution of the translational equations should be done by the analog computer. Because this division cannot completely isolate the rotational equations to the analog computer and the translational equations to the digital computer, several problems arise.

The problems

In the hybrid computer solution of differential equations several sources of error occur that are either not present or are easily compensated for in all digital or all analog computation. One of these sources of error, as reported by Miura and Iwata,[3] is the delay introduced into a closed loop because of digital computation. The length of this delay is the time from an analog voltage being sampled and transferred to the digital computer until all computations using that value are completed and the results returned to the analog computer. This source of error has a counter-part in an all digital simulation but there are well known methods of reducing the error, in the all digital case. Some of these methods are directly applicable to hybrid computation, some can be used in modified form and some cannot be used at all. The most notable of the last group is the Runge-Kutta method. To use this method for hybrid computation would require the analog computer to make several trial solutions at the same instant of time which cannot be done. A number of papers have been written discussing various methods from the first two groups but most are only effective for linear problems.

Another source of error in a hybrid computer is data reconstruction following digital to analog conversion. This source of error, as well as some others, is discussed by Karplus.[4] Theoretically, the output of a digital to analog converter is a string of impulses that are passed through some form of holding circuit. It is normally considered that reduction of the error by using any holding method more complicated than a simple zero order hold is not worth the problems created. This contention is not disputed but this paper will describe a method by which the effect of a higher order hold can be achieved without the actual hold circuit being present. One of the problems of using a zero order hold, which holds the value of the voltage constant between conversions, is that if the variable converted is a derivative to the integrated on the analog computer, which is often the case, especially in vehicle simulators, the integral can be determined algebraically at each sampling time. This means that although the integra-

tion is done by the analog computer, the result is the same as if it were done by the digital computer using the most elemental method of integration, i.e., rectangular or Euler. This being the case, there is some question as to the utility of using a hybrid computer for solving differential equations or at least for solving the equations of motion for a vehicle simulator.

A solution

The suggested solution, which is similar in some respects to that proposed by Gelman,[5] to the problem of derivative functions being represented by a series of step functions is to rewrite the equations so that every loop can be closed without going through the digital computer. As an example, to check the method, the equation

$$\frac{d^2y}{dt^2} = -\,ay \mid y \mid$$

was chosen because of several factors. These factors are, first, it is nonlinear; second, it has a solution that is periodic with constant amplitude and the period is functionally related to the amplitude; third, the period and amplitude can be determined analytically so errors can be checked; and last, it can be mechanized on an iterative differential analyzer simulating a hybrid computer. Actually this equation can be solved entirely by the analog computer, but the game was played assuming that the absolute value could not be found on the analog and thus the digital computer (or simulated digital computer) was needed.

The general form of this equation is

$$\frac{d^2y}{dt^2} = f(y, \frac{dy}{dt}, t)$$

or in state quation form it is

$$\frac{dx}{dt} = f\,(y,x,t)$$

$$\frac{dy}{dt} = x$$

If x and t do not appear explicitly in the function f, as in the example, the simplest way to rewrite the equation is to divide and multiply by y and mechanize it such that the function divided by y is generated by the digital computer and the multiplication by y is done by the analog. This mechanization might have some computation problems if y goes to zero but can be overcome if f also goes to zero for some y. To do this, divide and multiply by y plus a constant, such that y plus the constant goes to zero when f goes to zero and define zero over zero as zero.

This mechanization will not completely eliminate the problems of the time delay or the use of the zero order hold. However the outputs of the analog integrators can no longer be computed algebraically at a sampling instant from the conditions at the previous sampling instant. Instead, a linear differential equation must be solved. The result of this manipulation, therefore, is to change a nonlinear differential equation to a linear equation with parameters that change at every sampling instant so as to approximate the original equation. This is also approximately the result of using a higher order hold in the data conversion equipment.

Nothing in the foregoing discussion does anything directly to overcome the time delay problem. The example will be used to describe the attack on this problem rather than a general case. Assume that the analog to digital converter, ADC, and digital to analog converter, DAC, operations occur simultaneously and repeat with a period Δt, then the example equation as mechanized becomes

$$\frac{d^2y(t)}{dt^2} = -a \mid y(t_1 - \Delta t) \mid y(t)$$

for $t_1 \leq t < t_1 + \Delta t$. This means that the output of the DAC is delayed from the true function by one to two sampling periods, Δt. To overcome this delay, either the function that is transferred by the DAC, $-a \mid y\,(t_1 - \Delta t) \mid$, or the function transferred by the ADC, y, can be predicted ahead one and one half periods using a truncated Taylor Series. The function y was chosen since the Taylor Series requires derivatives of the function and they are present in the analog computer. Therefore,

$$y_p\,(t_1 + \frac{\Delta}{2}t) = y\,(t_1 - \Delta t) + \frac{dy}{dt}\,(t_1 - \Delta t)$$

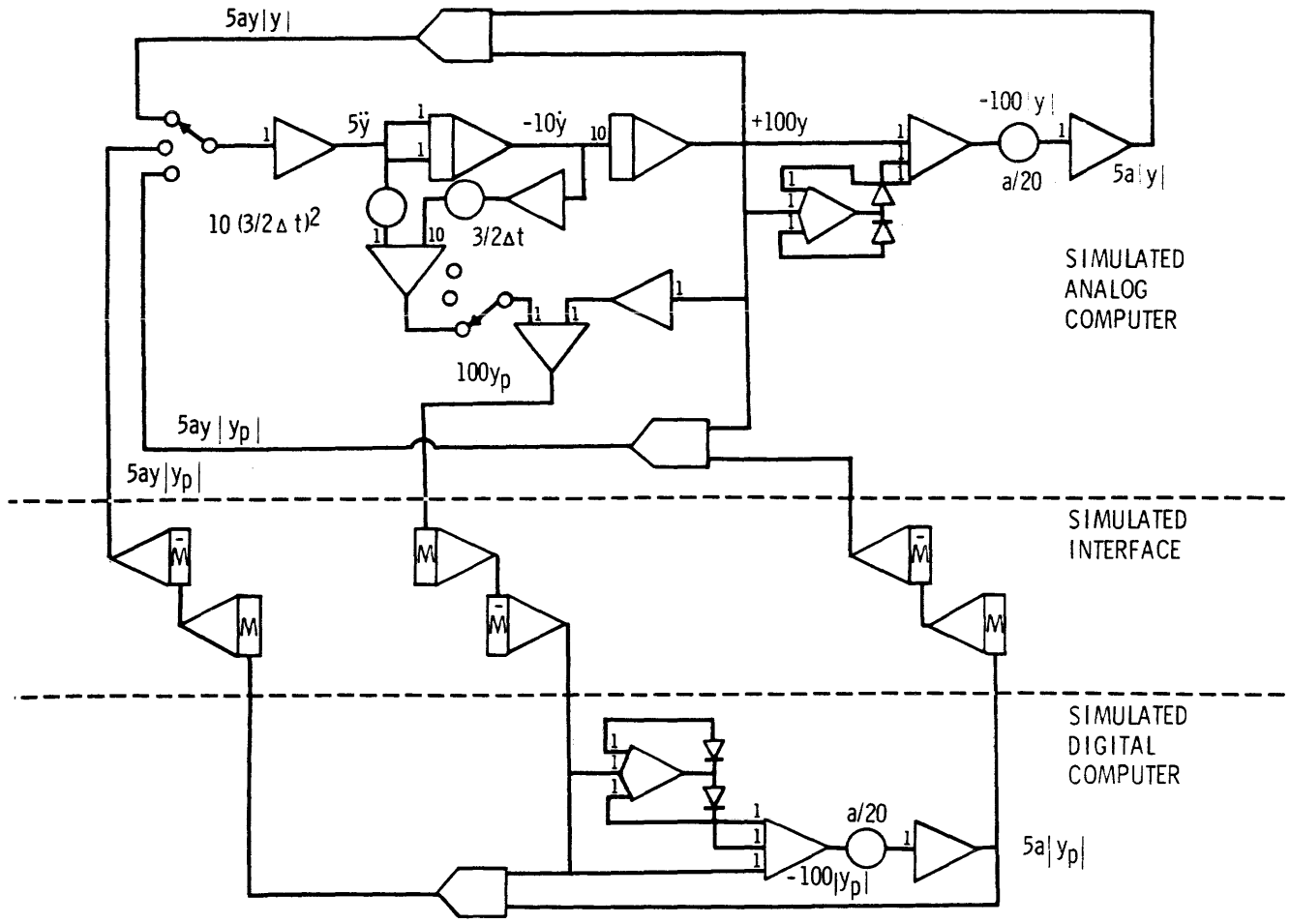


Figure 1 – Computer diagram for simulated hybrid solution of $y = -a\,y\,|\,y\,|$ by trial methods

$$(\frac{3}{2}\Delta t) + \frac{d^2y}{dt^2}(t_1 - \Delta t)\,\frac{2(\frac{3}{2}\Delta t)^2}{2}$$

If higher derivatives of the function to be predicted are available in the analog computer without differentiating they can be used to add more terms to the series.

The example equation was mechanized on an EASE 2100 Iterative Differential Analyzer as shown in the circuit diagram, Figure 1. The function switches allow for five different methods of solution. These are: all analog; hybrid with no compensation; hybrid with prediction only; hybrid with closed analog loop; and hybrid with both prediction and closed analog loop. Figure 3 shows the results of these five methods. The parameters and initial conditions were

$a = 16$, $y(o) = 0$, $\frac{dy}{dt}(o) = 2.8$. The Δt for hybrid with no compensation and hybrid with closed analog loop was .015 seconds and for the other two hybrid methods .15 seconds.

Although the solution using both compensations has a smaller change in amplitude with time than the solutions using the other methods, the error is noticeable and is dependent upon the sampling period. This last fact is particularly serious since the error tends to increase as the sampling period decreases.

To overcome this problem a further refinement was tried. Instead of dividing and multiplying f(y) by y, a truncated Taylor Series expansion of f(y) about y_p was tried. The example equation, using only two terms of the series, becomes

$$\frac{d^2y}{dt^2} = -a\,|\,y_p\,|\,y_p - 2a\,|\,y_p\,|\,(y - y_p)$$

since the second term of the series is $\frac{df(y_p)}{dy_p}(y - y_p)$ and the derivative of $y\,|\,y\,|$ with respect to y is $2\,|\,y\,|$.

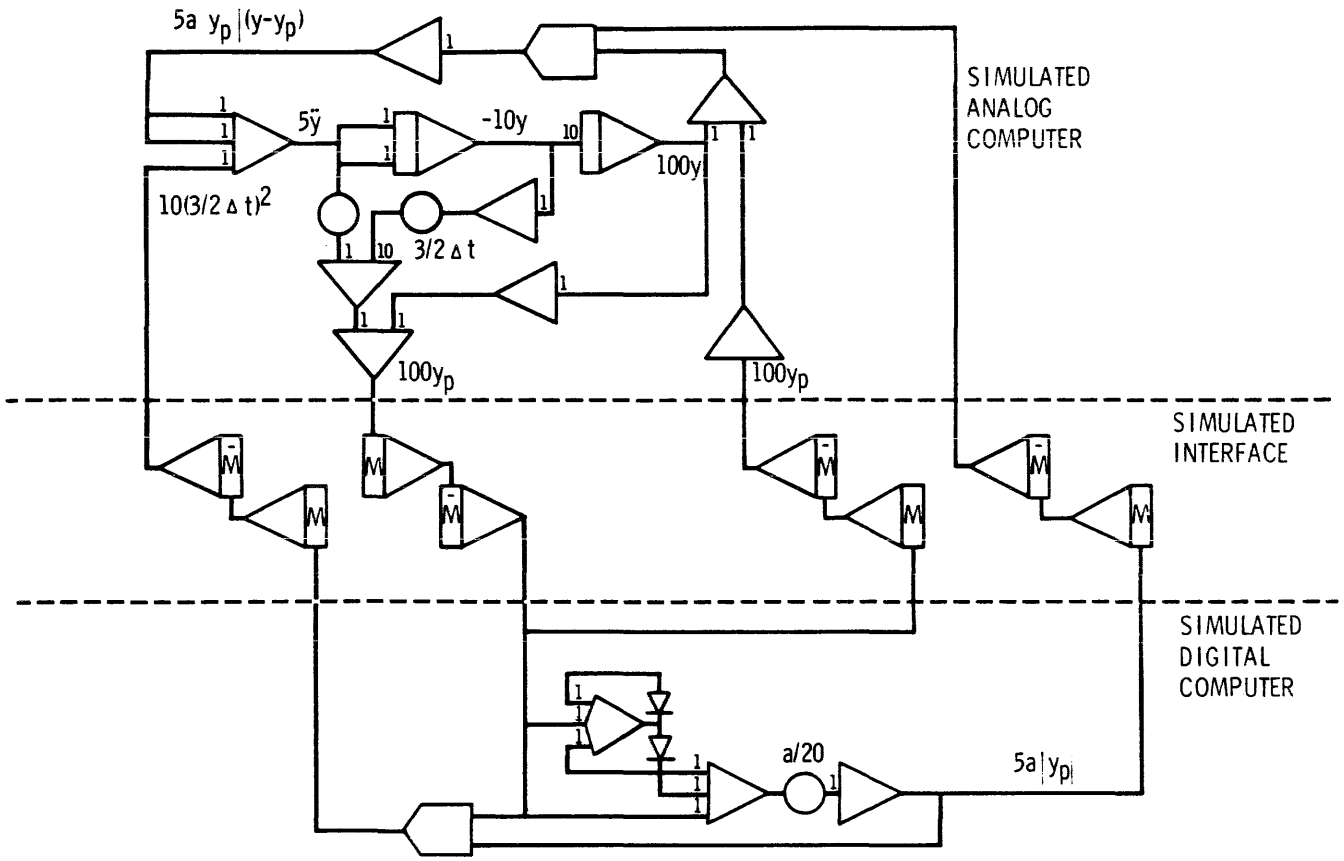


Figure 2—Computer diagram for simulated hybrid solution of $y = -a\,y\,|\,y\,|$ by final method

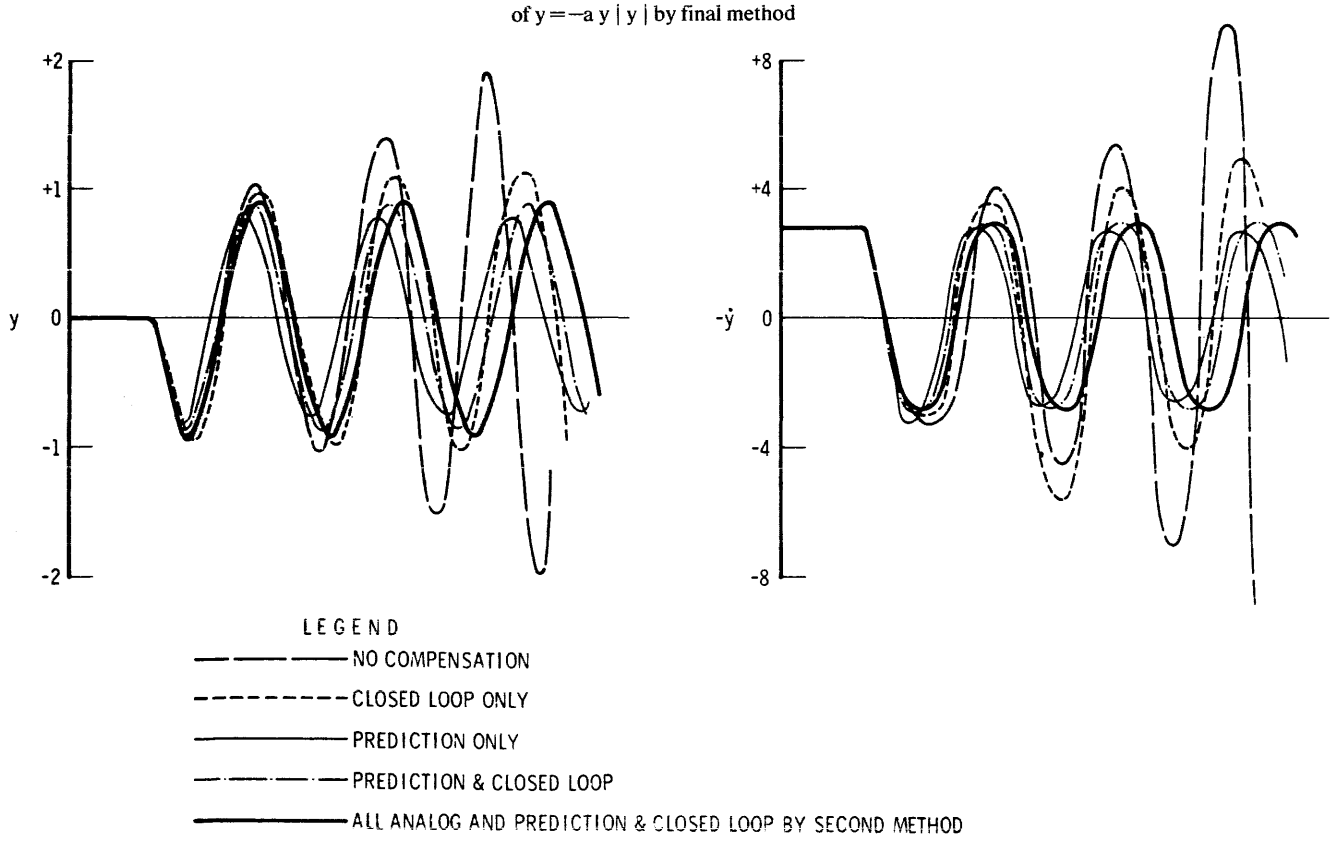


Figure 3—Solutions to the equation $y = -a\,y\,|\,y\,|$ by all methods

This equation, when mechanized on the hybrid computer, has the desired closed analog loop.

The computer diagram, again using the EASE to simulate a hybrid computer, is shown in Figure 2 and the results using the same parameters as before are shown in Figure 3 also. Other runs were taken using different sampling period, and for every run with a sample period shorter than the one shown, the solutions were identical as near as could be determined by eye. The runs made with larger sampling period showed a convergence so that in no case did the solution increase without bound.

Using the series expansion on f(y) not only reduces the error in solution to a negligible amount but also eliminates the possibility of division by zero and allows the method to be used for the general case where the derivative is a function of more than one variable. In this general case of a function of the form

$$\frac{dx_i}{dt} = f(x_1,x_2 \ldots x_n,t)$$

assume x_i,x_j,x_k,x_m, and t are generated by the analog computer and the rest of the x's are generated by the digital computer. Then x_{ip}, x_{jp}, x_{kp} and x_{mp} are generated using as many terms of a Taylor Series as there are derivatives available on the analog computer. The prediction of t is simply $t_p = t_l + 3/2\Delta t$. These variables are transferred to the digital computer and used to generate the functions $f(x_1,x_2,\ldots,x_{ip},\ldots, x_{jp},\ldots,x_{kp}, x_{mp},\ldots,x_n,t_p)$,

$$\frac{\delta f}{\delta x_{ip}}, \quad \frac{\delta f}{\delta x_{jp}}, \quad \frac{\delta f}{\delta x_{kp}}, \quad \frac{\delta f}{\delta x_{mp}}, \quad \text{and} \quad \frac{\delta f}{\delta t_p}.$$

These functions are then transferred back to the analog and combined to form the derivative

$$\frac{dx_i}{dt} = f + \frac{\delta f}{\delta x_{ip}}(x_i - x_{ip}) + \frac{\delta f}{\delta x_{jp}}(x_j - x_{jp})$$

$$+ \ldots + \frac{\delta f}{dt_p}(t - t_p)$$

where the predicted variables were either maintained on the analog computer by using memory integrators or were transferred back from the digital computer.

There are a few more facts and refinements of the general method that are of interest. To further check the method, the example was programmed for an IBM 7094 using MIMIC to simulate both parts of a hybrid computer. The program also contained a non-hybrid solution to use as a check. Several runs were made using different sampling intervals and the error after a given short period of time was compared for each run. Thinking of this hybrid integration as a numerical method, the order of the method was computed from these errors and was found to be fourth order. This leads to the thought that this method might also have an application in all digital simulations to reduce the machine time.

This hybrid technique is not without disadvantages. For one thing, it requires the use of more digital-to-analog converters than other methods. This is because the partial derivatives of the functions transferred to the analog must be transferred as well as the functions. Also the computation in the digital computer is somewhat more complicated because of the necessity to generate these partial derivatives. This is reduced, however, if the digital computation involves table look-ups for function generation. Digital table look-up and interpolation methods use the slopes (or partial derivatives) to perform the interpolation so they will be available for use by the analog computer.

Another problem is the computation of initial conditions for the variables transferred from the digital computer to the analog computer. Since these variables are functions of the predicted variables produced by the analog which in turn are functions of the original variables, an iteration process must be performed. This iteration takes place while the analog computer is in Initial Condition mode so time is not changing, hence the prediction required is only a half step instead of a step and a half. This necessitates extra analog circuitry and some switching logic.

Flight simulator mechanization

The simulator described is being developed as a general purpose six degree of freedom simulation applicable to missles, airplanes, or spacecraft; using a SDS 9300 digital computer and an EASE 2100 analog computer. The initial application is a short range guided missile and the description is based on this application.

The aerodynamic forces and moments for the missile and the partial derivatives of the forces and moments with respect to the aerodynamic attitude angles, α and β, fin deflection angles, δ_1, δ_2 and δ_3 and body rotational rates, P, Q, and R, are evaluated by the digital computer at the predicted sampling time. These values, including α, β and the δ's, are transferred through the interface to the analog computer where they are recombined into the Taylor Series approximation of the continuous functions. The moments are combined with the moments of inertia and body angular rates to produce total body angular accelerations. These are integrated to give the body rates and also transformed by the Euler

angles to produce the angular accelerations of the Euler angles for prediction. The body rates are transformed by the Euler angles to produce the Euler angle rates, which are integrated to give the angles. The Euler angles, rates, and accelerations are combined, using Taylor Series, to produce predicted Euler angles. The sines and cosines of the predicted Euler angles are generated and transferred through the interface to the digital computer to be used to compute the predicted α and β.

The aerodynamic forces and the thrust are converted to linear acclerations and transformed from body axes components to earth axes components. The accelerations are integrated to produce the changes in velocity due to external forces. This integration uses an incremental or "hybrid integrator" circuit similar to that described by Wait,[6] which operates until a fixed small increment of a velocity component has been accumulated and then generates an interrupt signal for the digital computer. The digital computer causes the integrator amplifier for that component to be reset to zero and adds the increment to the velocity component stored in digital memory and transfers the sum back to the analog computer. By using this circuit a more accurate solution can be achieved than by normal analog integration. The increments of velocity components and the linear accelerations are combined, using Taylor Series, to produce predicted velocity increments which are transferred through the interface to the digital computer.

The part of the velocity components returned from the digital computer are added to the increments to form total velocity components as continuous functions. These are transformed from earth axes back to body axes by the Euler angles and then transformed from rectangular to polar form yielding α and β as continuous functions. These are needed for the reconstruction of the continuous forces and moments.

The flight control sub-system of the missile is simulated on the analog computer. It receives guidance commands in the form of commanded Euler angles from the digital computer and compares them with the actual Euler angles to produce an error. This error is used to determine the appropriate fin deflection angles commands. The simulated fin deflection actuators produce the actual deflection rates and angles. These are combined to give the predicted fin deflection angles that are transferred to the digital computer.

The digital computations are performed in the following order. At the beginning of each sampling period or frame, the ADC channels are sampled and their values converted and stored. All of these variables are the predicted versions of the corresponding analog variables so that there is a frame and one half lead at this instant. The rest of the variables stored in the digital computer at this time are one half frame ahead. The digital computer now integrates over one frame the equations defining the change in velocity components due to gravity and the rotating coordinate system, i.e. earth axes, and the equations defining the translational position of the vehicle. Since this integration requires a certain amount of computation time, all of the variables in the digital that are involved in the translation equations of motion now have somewhere between one and one and a half frame lead over the analog computer. The velocity components are now transformed by the predicted Euler angle to body aces and converted to polar form, resulting in predicted α, β, and total relative vehicle velocity. The atmospheric properties are computed and the guidance equations are solved.

At this time, which is less than halfway through the frame all variables in the digital computer except the aerodynamic coefficients are updated to approximately one frame ahead of the analog computer. These variables include those for determining vehicle configuration changes such as thrust termination or staging or for determining the end of the run.

The tests are made and the time at which a change, if any, is to occur is determined by linear interpolation and the frame timer is set so as to stop the analog at the time of the change. The digital computer then goes back and recomputes the integrals up to the time of change. The reason for stopping the analog computer for just a configuration change is so that new data specifying that change can be read into the digital computer.

If no change conditions were met, the digital computer does the table look-ups of the aerodynamic coefficients as described previously and then waits until the fixed amount of time allowed for the frame has elapsed. The DAC values are then converted and after a very short delay to allow the transients on the analog computer due to the step changes in DAC outputs to settle, the ADC operation is done again and the next frame starts.

CONCLUSION

The six degree of freedom simulator discussed is not yet fully operational so a complete analysis of the accuracy and speed cannot be given. However, a three degree of freedom (one rotational and two translational) pitch plane only version of the simulator has been programmed using DES-1, the SDS 9300 digital analog simulation language, to represent a hybrid computer. The results from this simulator

indicate that the frequency and damping of the rotational motion agrees to within an acceptable limit with that calculated from a linearized model and that the accuracy of trajectory computations is as good as that from an all digital simulation. The frame time necessary to achieve this was 1/10 to 1/12 of the period of the highest frequency appearing at the interface.

REFERENCES

1 S FIFER
Analog computation
McGraw-Hill Book Company Inc New York Vol IV 1961

2 R M HOWE
Coordinate systems for solving three dimensional flight equations
Tech Report WADCTN55-747 Wright Patterson AFB Dayton Ohio June 1956

3 T MIURA J IWATA
Effects of digital execution time in a hybrid computer
Proc 1963 FJCC pp 251-265 AFIPS Conf Proc Vol 24 Spartan Washington DC 1963

4 W J KARPLUS
Error analysis of hybrid computer systems simulation Vol 6 pp 120-136 February 1966

5 R GELMAN *Corrected inputs-*
a method for improved hybrid simulation
Proc 1963 FJCC pp 267-276 AFIPS Conf Proc Vol 24 Spartan Washington DC 1963

6 J V WAIT
A hybrid analog-digital differential analyzer system
Proc 1963 FJCC pp 277-293 AFIPS Conf Proc Vol 24 Spartan Washington DC 1963

Backward time analog computer solutions of optimum control problems

by MAX D. ANDERSON
Autonetics Division, North American Aviation
Anaheim, California
and
SOMESHWAR C. GUPTA
Arizona State University
Tempe, Arizona

INTRODUCTION

Optimum control problems can be solved by iterative computing methods. However, the techniques used for parameter searching are quite sophisticated; consequently, a hybrid computer or a large digital computer is required for their mechanization.[1,2,3,4,5,6,7,8] These computing systems are very expensive and are not readily available. Furthermore, the trend is towards larger hybrid and digital computer requirements. This situation could be greatly helped by finding simpler methods for solving these problems. One such method utilizing an analog computer is developed in this paper.

Statement of the problem

Let the dynamic system of order n be described in state variable notation by the following set of n simultaneous first order differential equations.

$$\dot{\bar{x}} = \underline{A}\bar{x}(t) + \underline{B}\bar{u}(t) = \bar{f}(\bar{x},\bar{u}), \qquad (1)$$

with initial conditions $\bar{x}(t_o)$, where $\bar{x} = (x_1,x_2, \ldots, x_n)$ is an nxl state vector, $\underline{A}$ is an nxn matrix, B is an nxr matrix, and $\bar{u} = (u_1,u_2, \ldots u_r)$ is the rxl control vector. The optimum control problem is to find an admissible control $\bar{u}(t)$ that will drive the system described by the equations (1) from the initial states $x(t_o)$ to some final states $\bar{x}(t_f)$, the latter of which may not be specified as a point, in such a manner that the criterion function

$$J = \int_{t_o}^{t_f} f_o(\bar{x},\bar{u})\,dt \qquad (2)$$

attains a minimum value. The problem of minimizing J can be transformed into the problem of minimizing a new coordinate in state space defined by the functional

$$x_o(t) = \int_{t_o}^{t} f_o(\bar{x},\bar{u})\,dt. \qquad (3)$$

Then

$$x_o(t_f) = J. \qquad (4)$$

Any admissible control $\bar{u}(t)$ that satisfies these requirements is called an *optimum control.* The final time, t_f, may be either fixed or arbitrary.

Methods have been developed for solving this problem using Pontryagin's Maximum Principle.[5,6,7,9] This principle reduces the optimization problem to a two point boundary value problem. A searching technique must then be used to find an acceptable set of adjoint initial conditions, $\bar{p}(t_o)$, that will enforce the desired final states, $\bar{x}(t_f)$, of the system. Furthermore, for each state variable, x_i, that is constrained, another parameter, λ_i, called a Lagrange multiplier, is required for a complete solution. These are in addition to the initial condition parameters, $\bar{p}(t_o)$. Both sets of parameters, $\bar{p}(t_o)$ and $\bar{\lambda} = \lambda_1,\lambda_2, \ldots, \lambda_n)$, are presently being found by iterative searching techniques.[3,4,5,6,7,11] For example, in an n^{th} order system, there are n initial conditions in $\bar{p}(t_o)$ to be searched. If each state variable in $\bar{x}(t)$ is constrained, then there will also be n Lagrange multipliers to be found; hence, the total number of parameters to be searched will equal 2n.[10] On the other hand, saturation type constraints on the control variable $\bar{u}(t)$ do not require additional computations.

Unconstraining the terminal states

The overall computational requirements can be greatly simplified by taking advantage of the properties of the Maximum Principle stated in Theorems A.1 and

A.2, and by avoiding the use of Lagrange multipliers in solving the problem. This can be accomplished by removing the constraints on the state variables. If this cannot be done actually, then it must be done virtually; i.e., the problem must be made to appear as though the state variables are unconstrained. This can be done in many cases by transferring the constraints from the state variables to the criterion function. A method for handling constraints on the final states $\bar{x}(t_f)$ will now be developed. Let an error function of time be defined by

$$e(t) = \tfrac{1}{2}[(x_1(t) - x_1(t_f))^2 + (x_2(t) - x_2(t_f))^2 + \ldots + (x_n(t) - x_n(t_f))^2]. \quad (5)$$

By minimizing $e(t)$, $\bar{x}(t)$ is made to approach $\bar{x}(t_f)$ as t approaches t_f without being regarded as constrained. The problem with final state constraints can now be solved by making the error function in equation (5) a part of the original criterior function for the problem. This is done by defining a new criterion function by the new coordinate $x'_o(t)$, which is defined as a function of time by

$$x'_o(t) = e(t) + x_o(t) \quad (6)$$

where $x_0(t)$ is the original criterion function, and $e(e)$ is defined by equation (5). By writing the new criterion function as a function of time, it is easier to solve the problem when the final time, t_f, is arbitrary. By definition the original criterion function, $x_0(t)$, has an absolute minimum at some final time, t_f'; however, the required final states, $\bar{x}(t_f)$, may not occur at that time ($t_f \neq t_f'$ necessarily). Consequently, $x_o(t'_f)$ is not the function to be minimized; instead, the criterion function should be

$$x'_o(t_f) = e(t) + x_o(t)\Big|_{t = t_f} \quad (7)$$

to enforce the required final states. Since the system is controllable, it is possible to vary the time, t_f, at which $\bar{x}(t_f)$ is enforced. Control is limited only by constraints on $\bar{u}(t)$. Hence, it may be possible to make $x(t_f)$ occur at the time $x_o(t)$ has an absolute minimum. Furthermore, the error function $e(t)$ may be multiplied by a positive real number to make $x'_o(t_f)$ as close as desired to the minimum value of $x_o(t)$ with the constraints on $\bar{x}(t_f)$. Therefore, the optimum control problem can be solved as though $\bar{x}(t_f)$ were unconstrained, because a new criterion function has been defined by equation (6) to handle the condition that $\bar{x}(t_f)$ is constrained to some point in X.

Several classes of optimum control problems can now be modified and stated for the additional restriction that the final states are now constrained rather than free.

1. *Minimum energy problem.* In this problem the terminal states, $\bar{x}(t_f)$, are specified as a point. The new criterion function is of the form

$$x'_o(t) = e(t) + \int_{t_o}^{t} g(\bar{x},\bar{u},t)dt, \quad (8)$$

Where $g(\bar{x}\bar{u},t)$ is a continuous function, and $e(t)$ is an error function defined by equation (5). This functional must be minimum at time t_f. The problem is solved as though $\bar{x}(t_f)$ were free, with the specified final states, $\bar{x}(t_f)$, apeparing only in the new criterion function. Hence, no Lagrange multipliers will appear.

2. *Minimum fuel problem.* In this problem the terminal states, $\bar{x}(t_f)$, are specified as a point. The new criterion function is of the form

$$x'_o(t) = e(t) + \int_{t_o}^{t} \sum_{j=1}^{r} c_j |u_j| \, dt, \quad (9)$$

Where c_j is a constant, u_j is a component of the control variable $\bar{u}(t)$, for $j = 1,2, \ldots ,r$, and $e(t)$ is an error function defined by equation (5). This functional must be minimized at time t_f. Since the specified final states, $\bar{x}(t_f)$, appear only in the new criterion function, the problem is solved as though $\bar{x}(t_f)$ were free. Hence, no Lagrange multipliers will appear.

3. *Terminal control problem.* In this problem the final states, $\bar{x}(t_f)$, are also specified. The new criterion function is of the form

$$x'_o(t) = e(t_f) + h(\bar{x}(t_f)), \quad (10)$$

where h is a continuous function of $\bar{x}(t_f)$, and $e(t)$ is defined by equation (5). This function must be minimized at time t_f. Now the minimum error problem, with criterion function defined by equation (5), is, in fact, a particular terminal control problem. Hence, if $h(\bar{x}(t_f))$ has the form of equation (5), then $e(t)$ need not be added to form a new criterion function; i.e., $x_0'(t) = x_0(t)$ will hold. Since the constrained $\bar{x}(t_f)$ appears only in the criterion function, the problem is already in the proper form to be solved as though $\bar{x}(t_f)$ were not constrained. Therefore, no Lagrange multipliers will appear.

4. *Minimum time problem.* In this problem the final states, $\bar{x}(t_f)$, are also constrained to a point in X. The use of Lagrange multipliers can be avoided by converting the minimum time problem to the minimum error problem described above. Let a new criterion function, $e(t_f)$, be defined as a function of time by equation (5). Instead of finding a control $\bar{u}(t)$ that will drive the system from $\bar{x}(t_o)$ to $\bar{x}(t_f)$ in minimum time, the same problem can be solved by finding a control $\bar{u}(t)$ that will drive the system from $x(t_o)$ to any point in the Euclidean space X of state variables, in such a manner

that $e(t)$ is minimized to zero at some time $t > t_o$. Then minimum time is the smallest time, t_f, at which

$$e(t_f) = 0. \tag{11}$$

The result is that the required final states, $\bar{x}(t_f)$, will be enforced at that time, since $e(t)$ is a measure of the terminal error with time. By this method, the final states, $\bar{x}(t_f)$, can be considered as being unconstrained; in reality, the constraints have merely been shifted from the final states to the criterion function by defining $e(t)$. As a result, it is not necessary to introduce Lagrange multipliers and perform a computer search for these additional parameters.

Method for solving the problem backwards in time

A method for converting a problem with state variable constraints to one which apepars to be without constraints has been shown. This method allows the classes of problems without constraints on the state variables to be enlarged considerably. The motivation is that many of the actual optimum control problems fall within these classes. A method will now be presented which greatly simplifies the computational requirements for solving optimum control problems without constraints on the state variables.

Consider the properties of Pontryagin's Maximum Principle, stated in Appendix A, that hold for a system with free or unconstrained state variables. By free (unconstrained) state variables is meant that the state vector $\bar{x} = (x_1, x_2, \ldots, x_n)$ can be any point in the n-dimensional Euclidean space X. If $\bar{x}$ remains within some smooth manifold $S \leq X$ without imposing constraints, then the state variables also behave as though they were free. By requiring the state variables to be unconstrained, the hypothesis of Theorem A.1 is satisfied. For a linear system with additive control $\bar{u}(t)$ Theorem A.1 provides both a necessary and a sufficient condition for optimum control. In addition, this theorem provides final values for the adjoint variables; i.e.,

$$\bar{p}(t_f) = (-1,0,0,\ldots,0) \tag{12}$$

or

$$p_o(t_f) = -1 \text{ and } p_i(t_f) = 0, \text{ for } i = 1,2,\ldots,n.$$

With this additional information, brought about by the requirement that the state variables be unconstrained, a new computational procedure can be developed. Instead of solving an initial value problem with $\bar{x}(t_o)$ known and $\bar{p}(t_o)$ unknown, the problem can be solved as a final value problem with $\bar{x}(t_f)$ known or arbitrary and $\bar{p}(t_f) =) (-1,0,0,\ldots,0)$ known from Theorem A.1. But it is very difficult to solve final value problems, even with the help of a computer. Many computer runs are required to find one solution whose trajectory passes through the required final values $\bar{x}(t_f)$ and $\bar{p}(t_f)$. A better approach is to convert this final value problem into an initial value problem, since the latter problem is easier to solve. This is true because only one computer solution is required to uniquely solve an initial value problem. This is in contrast to the many computer solutions required for the iterative parameter searching techniques.

The problem can be solved as an initial value problem by solving it backwards in time; i.e., by starting the solution at time t_f and solving the problem backwards in time until time t_o. The initial conditions for this backwards time method are $\bar{x}(t_f)$ and $\bar{p}(t_f) = (-1,0,0,\ldots,0)$. To state this approach formally, let τ be the independent variable for the backwards time solution; i.e., let τ be time running backwards. The, during the time interval $[t_o, t_f]$, τ is defined by

$$\tau = t_f - t \tag{13}$$

where t_f is the final time, and t is real time. The backwards time solution begins with $\tau = 0$, which occurs at time $t = t_f$, and ends when $t = t_o$; i.e., when $\tau = t_f - t_o$. For the case $t_o = 0$, τ runs from 0 to t_f. By making the change of variable,

$$t = t_f - \tau, \tag{14}$$

the problem is converted from a final value problem to an initial value problem. The final states $\bar{x}(t_f)$ and $p(t_f)$, become the initial values for the backward time solution, and $\bar{x}(t_o)$ and $\bar{p}(t_o)$ become the final values upon which the backward time solution must terminate.

The method for solving an optimum control problem backwards in time can now be stated in three parts.

1. Write a program to solve the system equations
$$\dot{\bar{x}} = \underline{A}\bar{x}(t) + \underline{B}\bar{u}(t) \tag{15}$$
backwards in time. This can be accomplished by negating each input to every integrator or integration subroutine. The initial conditions are $\bar{x}(t_f)$.
2. Maximize the Hamiltonian H with respect to $\bar{u}$ mathematically, and program the resulting expression for $\bar{u}$, which is a function of $\bar{x}$ and p.
3. Write a program to solve the adjoint system equations
$$\dot{p}_i = -\frac{\partial H}{\partial x_i}, \text{ for } i = 1,2,\ldots,n \tag{16}$$
backwards in time, where the partial derivative of H with respect to x_i is taken by hand and the resulting equation programmed on the computer. The initial conditions are $\bar{p}(t_f = (-1,0,0,\ldots,0)$.

Note that the equation for H is *not* programmed on the computer. This method can be represented by the block diagram shown in Figure 1. There are now 2n equations in 2n unknowns with 2n initial conditions known; hence, there are no more initial conditions for which to search. Furthermore, no Lagrange multipliers have been used. Therefore, all parameter searching has been eliminated.

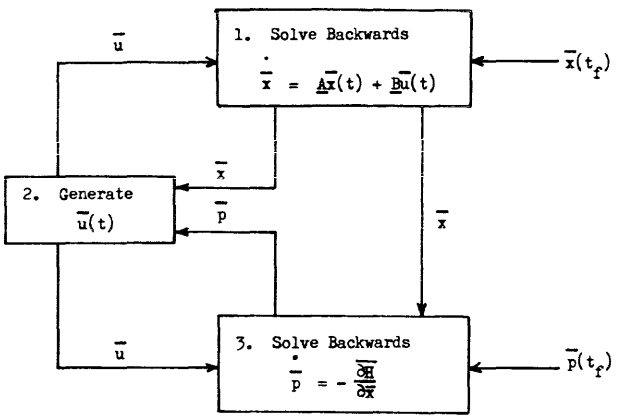


Figure 1—Method for solving an optimum control problem backwards in time on a computer

Examples

Example 1: Minimum time frequency control of phase-locked loops

The problem is to minimize the time for a phase-locked loop to lock onto a step change in frequency.[12] The sine term represents the non-linearity of the phase detector. The system is described by the state differential equations

$$\dot{x}_1 = \sqrt{2\omega_n}\sin(-x_1) + x_2 - u,\ x_1(0) = 0 \quad (17)$$

$$\dot{x}_2 = \omega_n \sin(-x_1),\ x_2\,0 = -\Delta\omega \quad (18)$$

n

Find the control u(t) that will drive the system from its initial states to the final states $x_1(t_f) = x_2(t_f) = 0$ in minimum time, where $|u| \leq K$.

The criterion function of minimum time is converted to a minimum error function defined as the new coordinate

$$\dot{x}_o(t) = \tfrac{1}{2}[(x_1(t) - x_1(t_f))^2 + (x_2(t) - x_2(t_f))^2]. \quad (19)$$

Then minimum time is the shortest time, t_f, at which $x_o(t_f)$ goes to zero.

The Hamiltonian is

$$H = p_o\dot{x}_o + p_1\dot{x}_1 + p_2\dot{x}_2$$
$$= (p_1 - x_1[\sqrt{2}\omega_n \sin(-x_1) + x_2 - u] + (p_2 - x_2)\omega^2_n \sin(-x_1), \quad (20)$$

since $p_o = -1$ by Theorem A.1 and $x_1(t_f) = 0$. H is maximum with respect to u if

$$u = K \text{ sgn } (x_1 - p_1). \quad (21)$$

The adjoint equations are

$$\dot{p}_1 = [\sqrt{2}\,\omega_n(p_1 - x_1) + \omega^2_n\,(p_2 - x_2)]\cos(-x_1) + \sqrt{2\omega_n}\,\sin(-x_1) + x_2 - u \quad (22)$$

$$\dot{p}_2 = x_1 - p_1 + \omega^2_n \sin(-x_1). \quad (23)$$

To solve this problem backwards in time, the following equations must be programmed on the computer.

$$\dot{x}_1 = \sqrt{2}\,\omega_n \sin x_1 - x_2 + u, \qquad x_1(t_f) = 0$$
$$\dot{x}_2 = \omega^2_n \sin x_1, \qquad x_2(t_f) = 0$$
$$u = K \text{ sgn}(x_1 - p_1)$$
$$\dot{p}_1 = -[\sqrt{2}\,\omega_n(p_1 - x_1) + \omega^2_n(p_2 - x_2)]\cos x_1 + \sqrt{2}\,\omega_n \sin x_1 - x_2 + u, \qquad p_1(t_f) = 0$$
$$\dot{p}_2 = -x_1 + p_1 + \omega^2_n \sin x_1, \qquad p_2(t_f) = 0$$

An analog computer solution of these equations is shown in Figure 2. These results agree with those obtained by Sanneman and Gupta.[12] A small perturbation of the adjoint initial conditions to the values $p_1(t_f) = 0.02$ and $p_2(t_f) = 0.03$ produced an accurate solution. This indicates that analog computer errors are on the order of 0.3 percent of full scale. The problem was time scaled to let one second real time equal 10 seconds computer time. This reduced the sensitivity of the problem and allowed the computer to give a more accurate solution.

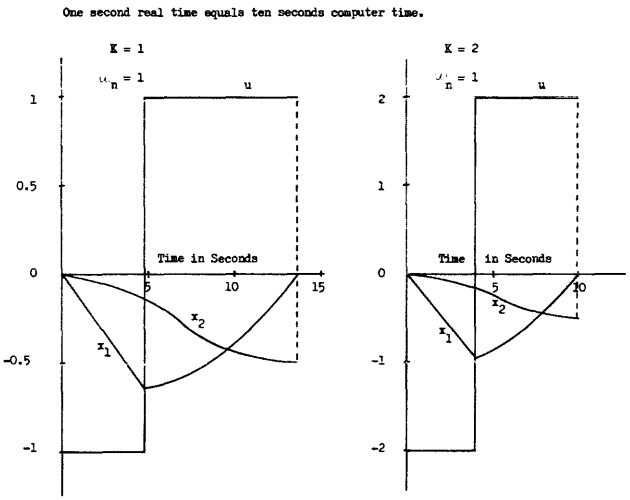


Figure 2—Solution of second order nonlinear phase-locked loop problem (Time Backwards)

Example 2: Third order nonlinear minimum Time problem

The purpose of this example is to show how much the difficulty of obtaining a solution increases in going from a second order to a third order nonlinear problem.

Let the system be described by the state differential equations

$$\dot{x}_1 = x_2, \qquad x_1(0) = 0.8 \quad (24)$$

$$\dot{x}_2 = x_3, \qquad x_2(0) = -0.8 \quad (25)$$

$$\dot{x}_3 = -ax_3 - x_2|x_2| + u, \qquad x_3(0) = -6.4 \quad (26)$$

Find the control u(t) which will drive the system from its initial states to the final states $x_1(t_f) = x_2(t_f) = x_3(t_f) = 0$ in minimum time, where $|u| \leq 1$ and $a = 0.333$. The criterion function of minimum time is converted to a minimum error function defined as a new coordinate by

$$x_0(t) = \tfrac{1}{2}[(x_1(t) - x_1(t_f))^2 + (x_2(t) - x_2(t_f))^2 + (x_3(t) - x_3(t_f))^2]. \quad (27)$$

Then minimum time is the smallest time, t_f, at which $x_0(t_f) = 0$. Since $\bar{x}(t_f) = 0$

$$\begin{aligned} \dot{x}_0 &= x_1\dot{x}_1 + x_2\dot{x}_2 + x_3\dot{x}_3 \\ &= x_1x_2 + x_2x_3 - ax_3^2 - x_2|x_2|x_3 + x_3u. \end{aligned} \quad (28)$$

The Hamiltonian is

$$\begin{aligned} H &= p_0\dot{x}_0 + p_1\dot{x}_1 + p_2\dot{x}_2 + p_3\dot{x}_3 \\ &= -x_1x_2 - x_2x_3 + ax_3^2 + x_2|x_2|x_3 - x_3u \\ &\quad + p_1x_2 + p_2x_3 - ap_3x_3 - p_3x_2|x_2| + p_3u, \end{aligned} \quad (29)$$

since $p_0 = -1$ by Theorem A.1. H is maximum with respect to u if

$$u = \text{sgn}\ (p_3 - x_3). \quad (30)$$

The adjoint equations that must be solved are

$$\dot{p}_1 = x_2, \qquad p_1(t_f) = 0 \quad (31)$$

$$\dot{p}_2 = x_1 + x_3 - 2|x_2|x_3 - p_1 + 2p_3|x_2|, \quad p_2(t_f) = 0 \quad (32)$$

$$\dot{p}_3 = x_2 - 2ax_3 - x_2|x_2| + u - p_2 + ap_3, \quad (33)$$

To obtain a solution, equations (24,25,26) and (31, 32, 33) had to be solved backwards in time along with equation (30). This problem was solved on the analog computer, and the results are shown in Figure 3. No particular difficulties were encountered in obtaining a solution. Only one multiplier, two integrators, and one amplifier were required for the third order problem in addition to those required for the second order problem. This was insignificant as was the additional effort to perturb $\bar{p}(t_f) = \pm \bar{\epsilon}$ for one additional component. This problem was also slowed down by 10, by time scaling, to obtain less sensitive and more accurate results.

Nonlinear problems have been chosen for the examples in order to demonstrate the capability of this method to solve nonlinear problems with the analog computer. On the contrary, it has not been shown in this work that the Maximum Principle is a sufficient condition for optimal control of nonlinear systems. It turns out, however, that the sufficiency condition holds for these examples. To show this, the notion of normality relative to X and the solution of the Hamiltonian-Jacobi partial differential equations are required. These have been worked out by Athans and Falb. [13]

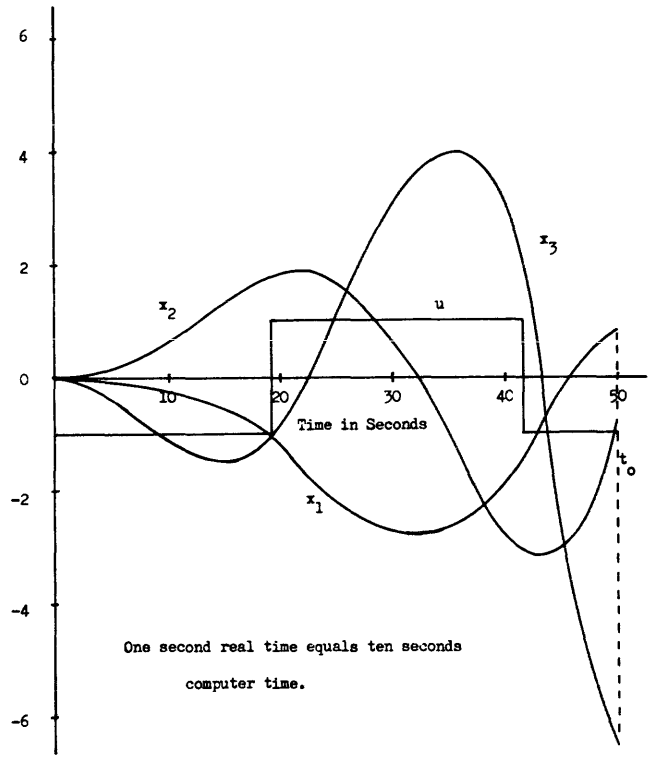


Figure 3—Solution of third order nonlinear problem (time backwards)

CONCLUSIONS

A simple method has been developed for solving optimum control problems by Pontryagin's Maximum Principle. The requirement for searching the adjoint initial conditions for an acceptable set and the use of Lagrange multipliers has been eliminated by solving the problem backwards in time.

A good criterion for evaluating a computing method is to note the increase in the complexity of obtaining a solution when going to higher order systems. The operating procedure does not change for the backward time method; only the computer program becomes more complex. In going from a second order to a third order system, the procedure for obtaining a solution remains the same, except that there is one more parameter, $p_3(t_f)$, to perturb by a small amount $\pm \epsilon_3$ to measure error and sensitivity. On the other hand, in the forward time mechanization, the computer time required to search for an acceptable $p(t_0)$ increases by an exponent equal to the order of the system; e.g., if m^2 seconds were required for a second order system to converge to a solution, then m^3 seconds are required for the convergence of a third order system. In addition, about 20 percent more analog hardware is required. Curves for evaluating these methods, based on several examples, are shown in Figure 4.

The capability of the backwards time method is also shown in its use as a design tool. If it is desired to

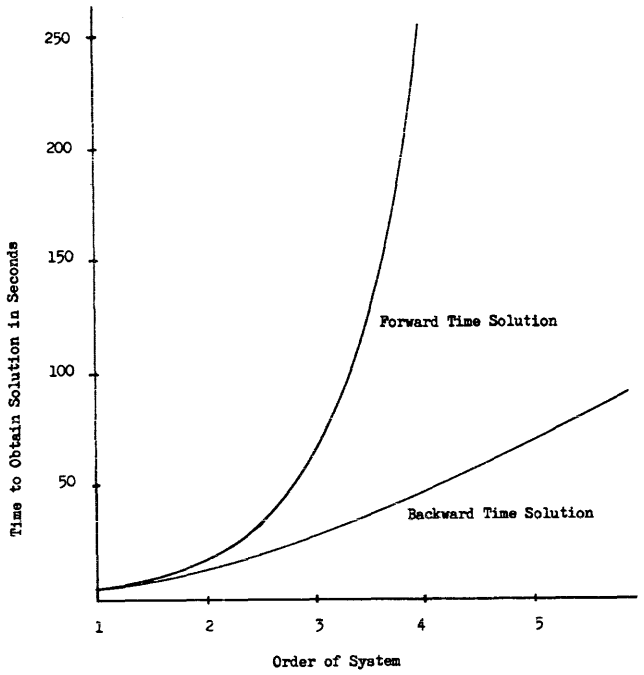


Figure 4—A comparison of the times required to obtain a computer solution

drive a system from a given initial state $\bar{x}(t_o)$ to a required final state $\bar{x}(t_f)$, it really doesn't matter whether the problem is solved forward or backwards in time. By solving the problem backwards in time from $x(t_f)$ with $u(t)$ constrained by $|u| \leq M$, the trajectories of $\bar{x}(t)$ are easily obtained, and the manner in which $\bar{x}(t_o)$ is approached can be observed. If the time $t_f - t_o$ is constrained, it can be seen immediately whether or not $\bar{x}(t)$ will move to $\bar{x}(t_o)$ in reasonable time. If not, then the information is available on how to change M to satisfy this requirement. Changing M is easily done on the computer. This approach is very difficult to achieve using the forward time method, because each time M is changed, a new set of initial states $\bar{p}(t_o)$ must be found by a searching procedure.

Appendix A

The following assumptions were made in order to solve the optimum control problem, stated in the body of the paper, by Pontrayagin's Maximum Principle.

1. The system is controlable; i.e., the states of the system can be changed from some initial states $\bar{x}(t_o)$ to some final states $\bar{x}(t_f)$ in a finite time $t_f - t_o$ by the application of some bounded control function $\bar{u}(t)$ over the time interval $[t_o,t_f]$. The state vector $x(t)$ is the solution to the state differential equations (1) with $\bar{x}(t_o)$ given. The state vector $\bar{x}(t)$ exists as a unique point $\bar{x}$ in the Euclidean space X of dimension n at each instant of time in the interval $[t_o,t_f]$.

2. There exists a control region U, which is a bounded subset of some r-dimensional vector space. The control variable $\bar{u}(t) \epsilon U$ must be a bounded piecewise continuous vector function of time, with domain in $[t_o,t_f]$ and range in U. If $\bar{u}(t)$ satisfies these requirements, it is called an admissible control.

3. The state differential equations (1) are continuous in the variables $\bar{x}$ and $\bar{u}$ and continuously differentiable with respect to the components of $\bar{x}$. The integrand $f_o(x,u)$ of the criterion function is also defined and continuous together with its partial derivative with respect to $\bar{x}$.

4. The system described by equations (1) is autonomous.

5. The criterion function J, defined by equation (2), has at lease a local minimum at time t_f. This is true when the first variation of J is zero and the quadratic form of the second variation of J is positive definite.[13]

The following theorems can be stated for systems with unconstrained final states:[13]

Theorem A.1: Let $\bar{u}(t)$ be an admissible control which moves the system described by equations (1) from the fixed initial states $\bar{x}(t_o)$ to some unconstrained final states $\bar{x}(t_f)$ in X. In order that $\bar{u}(t)$ be an optimal control and $\bar{x}(t)$ the corresponding optimal trajectory, it is necessary that there exists a nonzero continuous vector function $\bar{p}t() = (p_o(t),p_1(t), \ldots, p_n(t))$, corresponding to $\bar{u}(t)$ and $\bar{x}(t)$, on the time interval $[t_o, t_f]$, such that

a. $\bar{p}(t)$ is a solution to the adjoint system of equations defined by

$$\dot{p}_i = -\frac{6H}{6x_i}, \text{ for } i = 0,1, \ldots ,n, \qquad (A.1)$$

where $H(\bar{p},\bar{x},\bar{u})$ is defined by

$$H(p,x,u) = \sum_{i=0}^{n} p_i x_i. \qquad (A.2)$$

b. The functional $H(p,x,u)$ has an absolute minimum with respect to $\bar{u}(t)$ on the interval $[t_o, t_f]$.

c. The maximum value of $H(\bar{p},\bar{x},\bar{u})$ with respect to $u(t)$ is zero on the interval $[t_o, t_f]$; i.e.,

$$\sup_{u \epsilon U} H(\bar{p},\bar{x},\bar{u}) = 0 \qquad (A.3)$$

d. At the final time, t_f,

$$\dot{p}(t_f) = (-1,0,0, \ldots , 0). \qquad (A.4)$$

If the system is linear and the control additive, as in equations (1) for $\underline{A}$ and $\underline{B}$ matrices with constant elements, then Theorem A.1 provides both a necessary and sufficient condition for optimum control.

Theorem A.2: Let the system be described by the

set of differential equations (1), where the controls u_j are subject to the constraints

$$|u_j| \leq M_j, \text{ for } j = 1,2,\ldots,r, \qquad (11)$$

for M_j a positive real number. If the system is moved from its initial states $\bar{x}(t_o)$ to some unconstrained final states $\bar{x}(t_f)$ in X in such a manner that the criterion function defined by equation (5) is minimized to zero, then the condition $H(\bar{p}(t_f), \bar{x}(t_f), \bar{u}(t_f)) = 0$ is redundant.

REFERENCES

1 GEORGE A. BEKEY *Optimization of multiparameter* systems by hybrid computer techniques Simulation February, 1964, pp. 19-32, and March 1964, pp. 21-29

2 GEORGE A. BEKEY and ROBERT B. McGHEE *Gradient methods for the optimization of dynamic system parameters by hybrid computation* Computing Methods in Optimization Problems, edited by A. V. Balakrishnan and L. W. Neustadt Academic Press, Proc of U.C.L.A. Conference January 30, 1964, pp. 305-327

3 EDWARD J. FADDEN and ELMER G. GILBERT *Computational aspects of the time-optimal control problem* Computing Methods in Optimization Problems, edited by A. V. Balakrishnan and L. W. Neustadt Academic press, 1964, pp. 167-192

5 O. R. HOWARD and Z. V. REKASIDS *Error analysis with the maximum principle* IEEE Trans. on Automatic Control Vol. AC-9, No. 3 July, 1964, pp. 223-229

6 CHARLES H. KNAPP and PAUL A. FROST *Determination of optimal control and trajectories using the maximum principle in association with a gradient technique* Proc. of Joint Automatic Control Conf., 1964, pp. 222-225, also in IEEE Trans. on Automatic Control, Vol. AC-10, No. 2 April 1965, pp. 189-193.

7 R. L. MAYBACH *Optimal control by pontryagin on hybrid computer* IEEE Region Six Conference Record. April, 1965, also ACL Memo No. 119, Electrical Engineering Department, University of Arizona, April 27, 1966

8 BERNARD PAIEWONSKY, PETER WOODROW, WALTER BRUNNER, and PETER HALBERT *Synthesis of optimal controllers Using hybrid analog-digital computers* Computing Methods in Optimization Problems, edited by A. V. Balakrishnan and L. W. Neustadt, Academic Press, 1964, pp. 285-303

9 M. D. ANDERSON and S. C. GUPTA *Analog computer solution of a third-order pontryagin optimum control system* Simulation, Vol. 5, No. 4 October, 1965, pp. 258-263

10 DIANG-TSENG FAN *The continuous maximum principle* John Wiley & Sons 1966

11 JULIUS T. TOU *Modern control theory* New York: McGraw-Hill, 1964

12 R. W. SANNEMAN and S. C. GUPTA *Optimum strategies for minimum time frequency transitions in phase-locked loops* IEEE Trans. on Aerospace and Electronic systems, Vol. AES-2, No. 5 September, 1966, pp. 570-581

13 M. ATHANS and P. L. FALB *Optimal control* McGraw-Hill 1966

An application of hybrid curve generation–cartoon animation by electronic computers

by TAKEO MIURA
Hitachi Central Research Laboratory
Tokyo, Japan

and

JUNZO IWATA
Hitachi Electronics Co.
Tokyo, Japan

and

JUNJI TSUDA
Hitachi Central Research Laboratory
Tokyo, Japan

INTRODUCTION

Curve generation techniques are gaining importance in the field of computer graphics. Curve generators using either digital or analog techniques can be built. However it is extremely advantageous to use hybrid techniques by taking advantage of superior analog curve generation capability. This results in small storage requirements for a digital computer and a relatively fast display time.

In this paper, as an application of hybrid curve generation techniques and computer graphics, computer animation problems are dealt with.

Motion picture cartoons and other types of animation require extraordinarily great expenditures of labor, since each individual cartoon frame must be drawn by hand. These frames, each one of which incorporates only minute changes, are then photographed one after another. We have recently developed two computing methods for producing animation: (1) Representing the picture in mathematical equations and moving it by switching the constants of the equations (analog computer method); and (2) having two frames drawn by an animator. The curve indicating the movement between these two frames is then read into the computer. According to the results calculated by the computer, animations between the two frames are machine-drawn (hybrid computer method). The second method offers more advantages from the practical point of view, because it is easier to draw a picture which is faithful to the artist's intention.

Analog computer method

Basic principles

Individual pictures in animated cartoons consist of a multitude of highly complicated lines. However, each individual section of the picture can be simulated by means of relatively simple curves. For example, a curve can be approximated by a series of parabolas. In most cases, each section consists of simple closed curves. Consequently, it is possible to take a circle as the basis of a closed curve, and then to modify it to make a desired pattern.

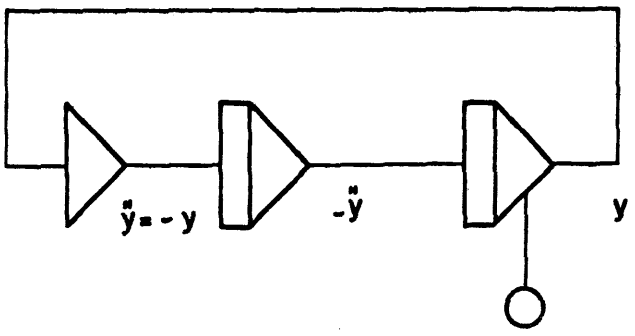


Figure 1 – Circle-test circuit

A circle can be generated by a so-called "circle-test circuit," shown in Figure 1. In an analog computer, the modification of a pictorial pattern can be performed simultaneously with the generation of the circle. Thus, it is possible to produce a picture by varying in succession the manner of modification as numerous circles are being produced repeatedly one after another. These are displayed on a cathode ray tube. If slight differences are introduced in the parameters in the transformation circuit for each one of these individual curves, then it will appear as if the picture is moving continuously.

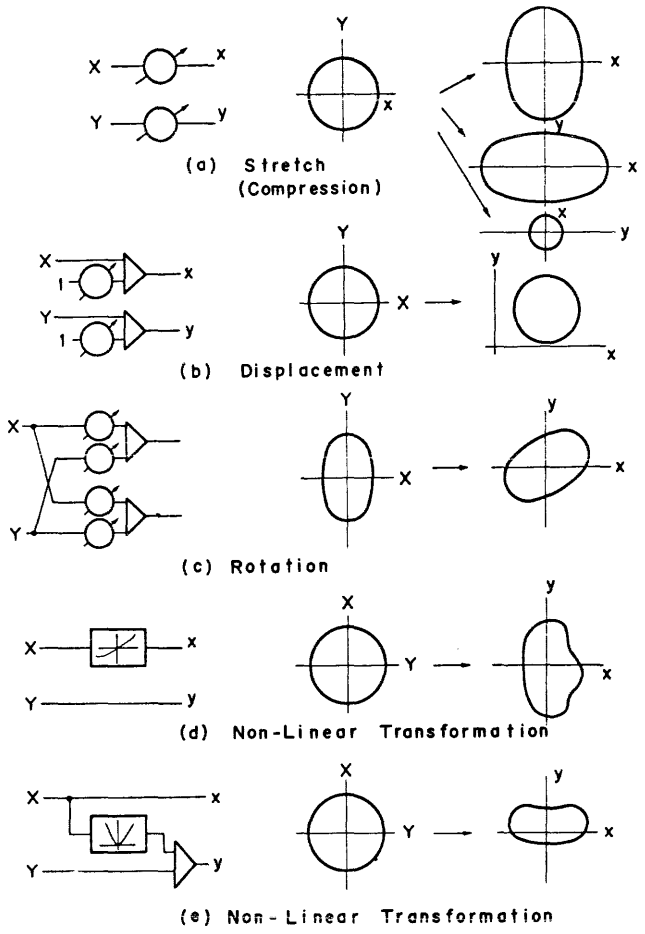


Figure 2 – Transformation methods

The chief means of modifying the basic pictorial patterns are as shown in Figure 2.

Although all of these are transformations of the coordinates, the procedures of (d) and (e) in Figure 2 have the effect of distorting the coordinate axis. Figure 3 shows how the coordinate grids are changed by these transformations. Thus, if one takes into consideration these transformations of the coordinate grids, one can anticipate the shape which a given pictorial pattern will have after its transformation. In this manner, one can modify the basic circle to produce a large number of desired shapes.

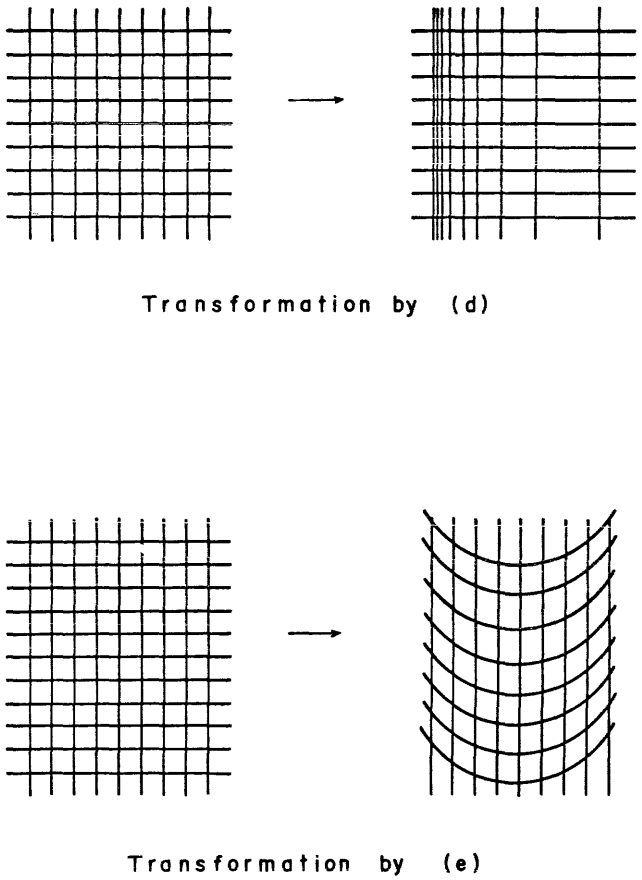
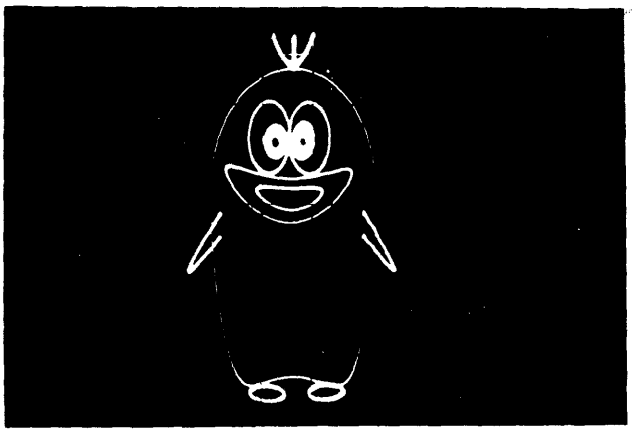


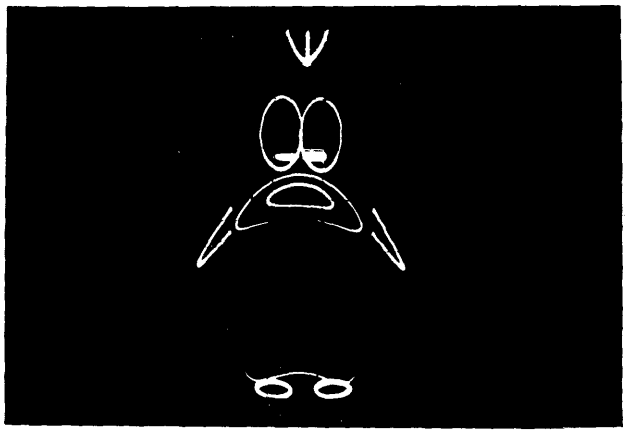
Figure 3 – Transformation of coordinate grid

Besides modifying circles, it is naturally possible also to begin with other differential equations. For example, one can use a damped oscillation circuit. If this is displayed on the phase plane, one will obtain logarithmic spirals. However, if the decrement is made smaller, it will be possible to black out the area inside the circle. If this process is discontinued before completion, one will obtain a circle with thick lines. Other differential equations for obtaining other solutions are also conceivable. However, just as in the case of modifications, if they are exceedingly complex, they will cease to be practical.

Examples of actual applications

Figure 4 is a picture of Oba-Q, the hero of a comic strip enjoying popularity in Japan. Figure 5 is a picture of rabbit driving a car. As for the former the truck, eyes, mouth, and legs consist of seven closed curves. The eyeballs are blacked out by spirals, and the hands and the hair are made by conics. The functions of movement, expansion and contraction, modification, etc., can be provided by several potentiometers in the computing circuit. This can be done





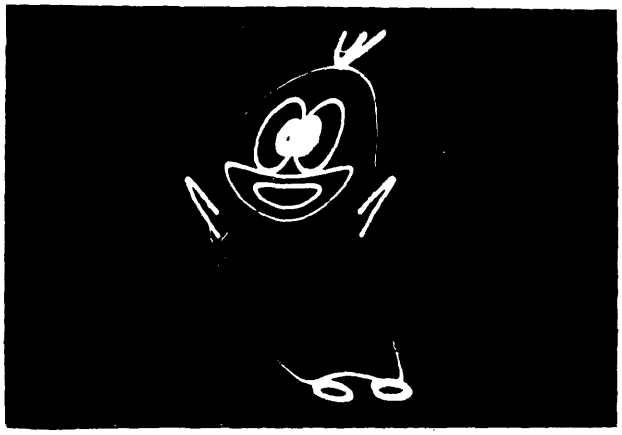


Figure 4 – Oba-Q displayed on a cathod ray tube

either manually or under the program control of the digital computer. In carrying out these experiments, we used the Hitachi ALS-2000 analog computer, which has an iteration rate of 3 KC.

Figure 5 – A rabbit driving a car

Problems of the analog computer method

This method has the advantage that the pictures produced on the cathode ray tube can be moved on real time. However, the method also has two important drawbacks. One is the fact that it is difficult to produce a picture which is faithful to the artist's intention. The second drawback is that in order to obtain complicated pictures the computing circuit itself will become complicated.

Hybrid computer method

General introduction

In this method an animator draws out by hand two separate frames of the cartoon. These two original drawings as well as the curves indicating the movements between the two, are read into the computer. Then the computer generates the intervening frames by interpolation.

A hybrid computer is used for the following reasons. First, input operations can be performed conveniently if the curves are read in an alog fashion. In addition, since the pictures must ultimately be drawn, analog techniques will be necessary in these sections. The curves were represented by means of the coordinates of representative points. Although in this method it is necessary to use a greater amount of data to represent the picture, it is possible to represent even highly complicated curves, and any picture conceived by the animator can be produced easily. The curves can also be modified or revised freely in any way.

Reading in the pictures

As is shown in Figure 6, a curve is represented approximately by suitable sampling points: P_1, $P_2 \ldots P_7$.

The following four types of points will be necessary:

a Initial point. This is the point where the curve begins. The pen of the XY recorder is lowered at this point.

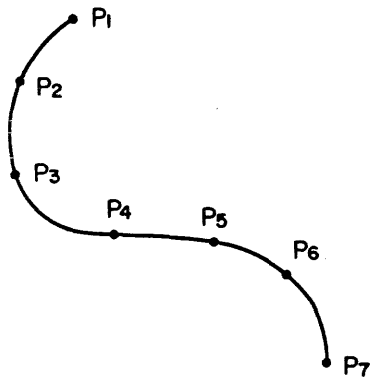


Figure 6—Representation of a curve by point coordinates

b Intermediate points. These are the intermediate points along the curve. They ought to be connected smoothly with the points preceding and following them.

c Break point. This is an intermediate point on a curve at which the curve bends in a different direction.

d Terminal point. This is the point where a curve ends. The pen of the XY recorder is raised at this point.

The pictures are read into the digital computer by means of a series of points, and are then memorized by the computer. Each point includes information concerning the position (the coordinates) and the type of point.

A special curve reader is used. This reader consists of a pen for position detection and a set of function switches. As the pen is moved along the curve, the coordinates of the points are detected electromagnetically and are transformed into digital quantities. The point coordinates are read into the computer by operating the switch. The type of point is distinguished by selecting the appropriate function switch.

Reproducing the pictures

There are various conceivable methods of reproducing a curve by joining together a series of points, for instance the method of joining the points by means of arcs or parabolas. We adopted a method of utilizing analog elements by which it was possible to draw these curves simply and conveniently.

As shown in Figure 7, if one specifies both the positions of each of the points and the tangents of the curves at each of the points, then it will be possible to reproduce the original curve with almost complete fidelity. However, a prerequisite for this is that the series of points must have been sampled with a sufficient degree of coarseness making possible a faithful representation of the original curve.

The following two processes must be performed:

1 The tangents must be calculated.

2 Then one must generate a smooth curve which will pass accurately through the given points and will connect with the given tangents.

The first process is performed by the digital computer, and the second by the analog elements.

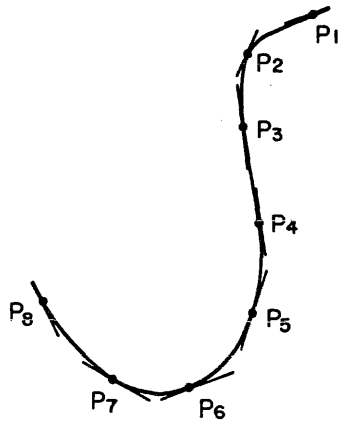


Figure 7—Curves are designated by the positions of the points and the directions of the tangents

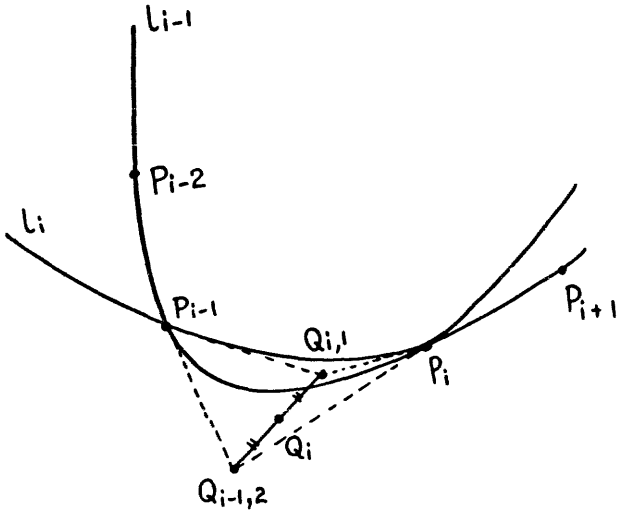


Figure 8—Calculation of tangents by parabolic approximation

Calculation of tangents

In order to calculate the tangents, the smooth curve passing through a given series of points must be expressed by equations. Here we adopted the parabolic approximation. Let us suppose that there are four points: P_{i-2}, P_{i-1}, P_i, and P_{i+1}, as shown in Fig. 8. First, we shall draw the parabola l_{i-1}, having an axis perpendicular to $P_{i-2}P_i$ and passing through the three points: P_{i-2}, P_{i-1}, and P_i. Let $Q_{i-1,2}$ be the point of intersection of the tangents of l_{i-1} at point P_{i-1} and point P_i. Next, we shall draw the parabola l_i, having an axis perpendicular to $P_{i-1}P_{i+1}$ and passing through the three points: P_{i-1}, P_i, and P_{i+1}. Let $Q_{i,1}$ be the point of intersection of the tangents of l_i at points

P_{i-1} and P_i. Taking into consideration Q_i, the central point between $Q_{i-1,2}$ and $Q_{i,1}$, we assumed that $P_{i-1}Q_i$ and P_iQ_i were the tangents at P_{i-1} and P_i of the approximate curve between points P_{i-1} and P_i. Since the tangents of the curve at each point will not strictly coincide before and after the point, the curve will therefore bend in a different direction at each point. However, if the points in the series have been spaced at suitable intervals, the differences in the tangential directions will be so small as to be negligible, and these variations will make no difference from the practical point of view.

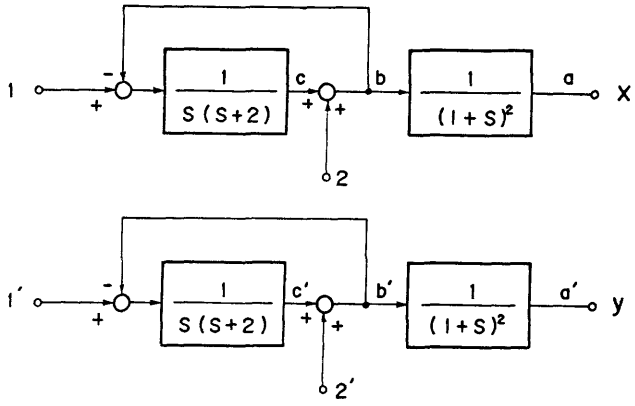


Figure 9—Block diagram for generation of a smooth curve

Generation of smooth curves

Having obtained the two points $P_1(x_1,y_1)$ and P_2 (x_2,y_2), as well as $Q(\xi,\eta)$, the point of intersection of tangents T_1 and T_2 at P_1 and P_2, we next draw a curve passing through these points and having the given tangents. This can be conveniently mechanized using the analog elements. Taking into consideration the dynamic characteristics of the XY recorder, we employed the circuit shown in block diagram form in Figure 9. Let us suppose that each input terminal is fed the values: $1 = x_1$, $1' = x$, $2 = X$ and $2' = Y$(X and Y are certain values), and that the circuit is in steady state. Then the outputs will be $x = x_1$ and $y = y_1$. Let us next change the input at 1 and 1′ to x_2 and y_2 respectively, and at the same time let us change the input at 2 and 2′ to $X + \xi - x_1$ and $Y + \eta - y_1$, respectively.

Then the response of the circuit will be:

$$x(t) = x_2 + [(x_1 - x_2)(1+t) + (\xi - x_2)\left(\frac{t^2}{2} + \frac{t^3}{6}\right)]\, e^{-t}$$

$$y(t) = y_2 + [(y_1 - y_2)(1+t) + (\eta - y_2)\left(\frac{t^2}{2} + \frac{t^3}{6}\right)]\, e^{-t}$$

Consequently:

$$x(0) = x_1 \quad x(\infty) = x_2 \quad \left.\frac{dy}{dx}\right|_{t=0} = \frac{\eta - y_1}{\xi - x_1}$$

$$y(0) = y_1 \quad x(\infty) = y_2 \quad \left.\frac{dy}{dx}\right|_{t=\infty} = \frac{\eta - y_2}{\xi - x_2}$$

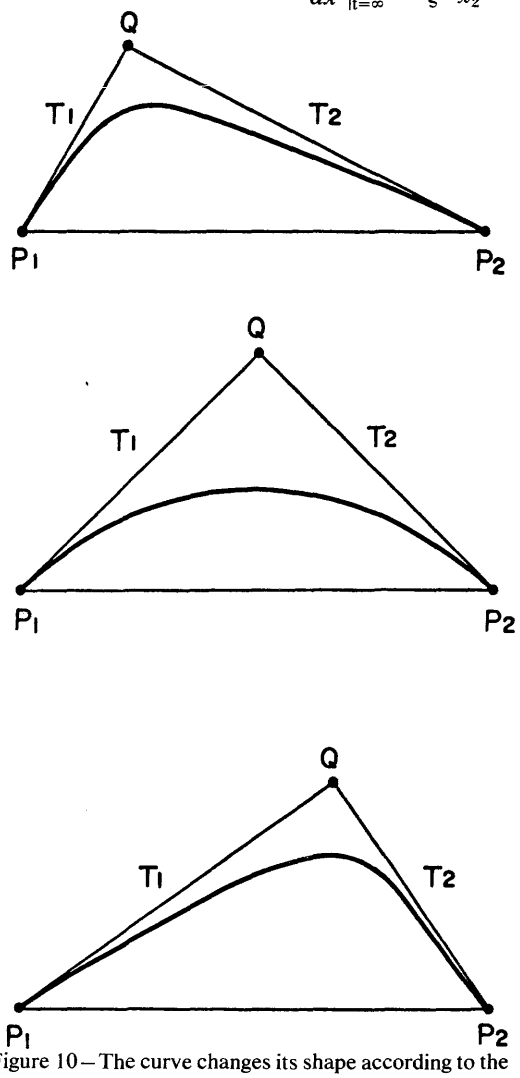


Figure 10—The curve changes its shape according to the tangents T_1 and T_2

Therefore, the resultant curve $l = (x(t), y(t))$ starts from P_1 (x_1,y_1), terminates at P_2 (x_2,y_2) and has the given tangents at P_1 and P_2. Curve l has a shape enclosed by the triangle QP_1P_2 and its shape changes depending upon the directions of tangents T_1 and T_2, as shown in Figure 10. This method can provide a seemingly natural curve and is quite adequate for the purpose of curve generation required here.

Interpolation of pictures between the two frames

There are two methods available now to provide a cartoon figure with a sequence of movement.

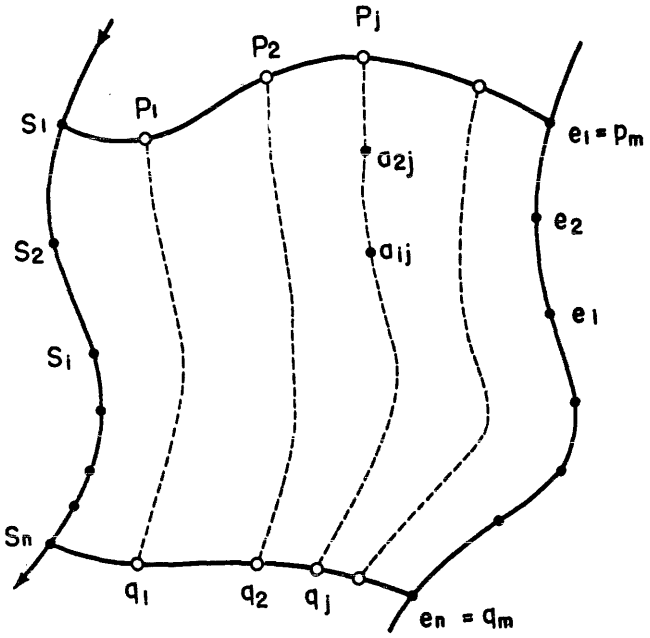


Figure 11 – Interpolation of Type I

Type I

In this method, the original curve is given, as well as the curve after it has been moved and/or modified. Furthermore, the sequence of movement is specified for representative points on the curve. Then the specified number of interpolation curves are prepared for the changes intervening between both curves.

Linear interpolation is adopted as the method of interpolation. That is, interpolation is performed by the fundamental operations of movement, rotation, and expansion and contraction.

Let us assume that the curve in the original drawing No. 1 is given by the series of points S_1, S_2 ..., S_n, and that the curve in the original drawing No. 2 is given by the series of points e_1, e_2 . . . , e_n (see Figure 11). Let us suppose that there is a one-to-one relationship between the points in both series, for instance, that point S_i will be at point e_i after it has completed its movement. Let us also suppose that points S_1 and S_n are chosen as representative points, and that the sequence of movements of each is given by points $P_1 \sim P_m$ and points $q_1 \sim q_m$ respectively.

If we wish to calculate A_{ij}, the ith point on the jth interpolation curve, we assume W^{si} to be the rotation matrix for superposing the vecotr S_nS_1 on vector q_jP_j

If the expansion rate is k^{sj}:

$$A^s{}_{ij} = k^{sj} W^{sj}(S_i - S_n) + S_i \quad \begin{pmatrix} i = 1, 2, \ldots, n \\ j = 1, 2, \ldots, m \end{pmatrix}$$

Next, let us assume W^{ej} to be the rotation matrix for superposing the vector e_ne_1 on vector q_jP_j. If the expansion rate is k^{ei}:

$$A^e{}_{ij} = K^{ei} W^{ei}(e^i - e^n) + q_j \quad \begin{pmatrix} i \,!\, 1, 2, \ldots, n \\ j \,!\, 1, 2, \ldots, m \end{pmatrix}$$

Finally, let us take the weighted mean of $A^s{}_{ij}$ and $A^e{}_{ij}$ to obtain the required point A_{ij}.
This will give:

$$A_{ij} = \{ (m - j)\, A^s{}_{ij} + j\, A^e{}_{ij} \}/m$$

Matrices $k^{sj}W^{sj}$ and $k^{ej}W^{ej}$ can each be calculated by the following equations:

$$k^{sj}W^{sj} = \frac{1}{(S_{1x} - S_{nx})^2 + (S_{1y} - S_{ny})^2} \begin{bmatrix} w^{sj}{}_1 & w^{sj}{}_2 \\ -w^{sj}{}_2 & w^{sj}{}_1 \end{bmatrix}$$

Here,

$$\begin{aligned} w^{wj}{}_1 &+ (S_{1x} - S_{nx})\,(P_{jx} - Q_{jx}) \\ &+ (S_{1y} - S_{ny})\,(P_{jy} - Q_{jy}) \\ w^{sj}{}_2 &= (S_{1y} - S_{ny})\,(P_{jx} - Q_{jx}) \\ &- (S_{1x} - S_{nx})\,(P_{jy} - Q_{jy}) \end{aligned}$$

And

$$k^{ej}W^{ej} = \frac{1}{(e_{1x} - e_{nx})^2 + (e_{1y} - e_{ny})^2} \begin{bmatrix} w^{ej}{}_1 & w^{ej}{}_2 \\ -w^{ej}{}_2 & w^{ej}{}_1 \end{bmatrix}$$

Here,

$$\begin{aligned} w^{ej}{}_1 &= (e_{1x} - e_{nx})\,(P_{ix} - Q_{jx}) + (e_{1y} - e_{ny})\,(P_{jy} - Q_{jy}) \\ w^{ej}{}_2 &= (e_{1y} - e_{ny})\,(P_{jx} - Q_{jx}) - (e_{1x} - e_{nx}\,(P_{jy} - Q_y) \end{aligned}$$

By means of this transformation, one can obtain the series of points for the group of curves changing continuously, in both their shapes and their positions, from curve S_1, $S_2 \ldots S_n$ to curve e_1, $e_2 \ldots e_n$.

Type II

In this case, one original drawing will suffice. The original curve is moved by a combination of movement, rotation, and expansion and contraction.

This type of transformation is depicted graphically in generalized form in Figure 12. The method of moving the curve is the following. The reference point S'_n is established at an appropriate point, and vector $S'_nS'_1$ is drawn from this reference point to a suitable length and in a suitable direction. This is taken as the reference vector. Next, if the curve is a *jth* transformation curve, two points: Q'_j and P'_j are given, and one determines the transformation relationship for shifting the reference vector to the vector $Q'_jP'_j$. The original curve is then moved using this relationship.

The following equation is used to calculate A_{ij}, the *ith* point on the *jth* interpolation curve:

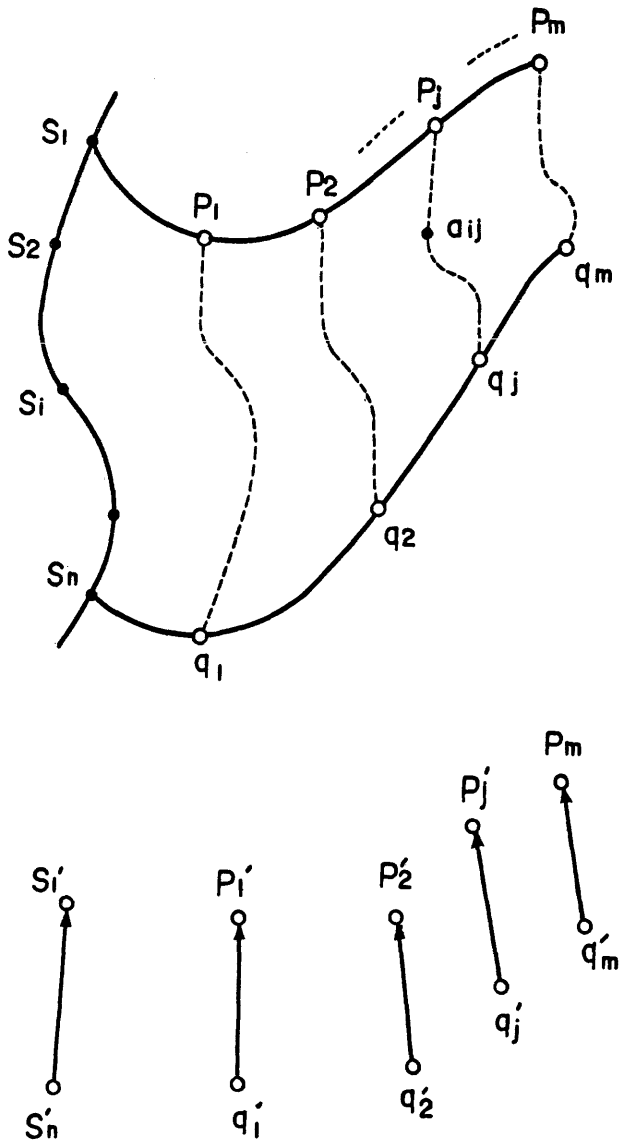


Figure 12 – Interpolation of Type II

$$A_{ij} = W^j (S_i - S_n) + Q'_j \quad \begin{pmatrix} i=1,2,\ldots,n \\ j=1,2,\ldots,m \end{pmatrix}$$

Here,

$$W^j = \frac{1}{(S'_{1x} - S'_{nx})^2 + (S'_{1y} - S'_{ny})^2} \begin{bmatrix} w^j_1 & w^j_2 \\ -w^j_2 & w^j_1 \end{bmatrix}$$

$$w^j_1 = (S'_{1x} - S'_{nx})(P'_{jx} - Q_{jx}) + (S'_{1y} - S'_{ny})(P'_{jy} - Q_{jy})$$
$$w^j_2 = (S'_{1y} - S'_{ny})(P'_{jx} - Q'_{jx}) - (S'_{1x} - S'_{nx})(P'_{jy} - Q'_{jy})$$

By means of this transformation, both the position and magnitude of the original curve can be transformed while maintaining a similar shape.

In designating the range within which the curves are moved, it does not matter what types of points the representative points are, nor does it matter what types of points are the points in between both of the representative points. For instance, if one wishes simply to move the picture as a whole, it will be enough merely to select the first and last points in the series of points as the representative points. The parts to be moved are not limited to a single place. Any number of places can be specified.

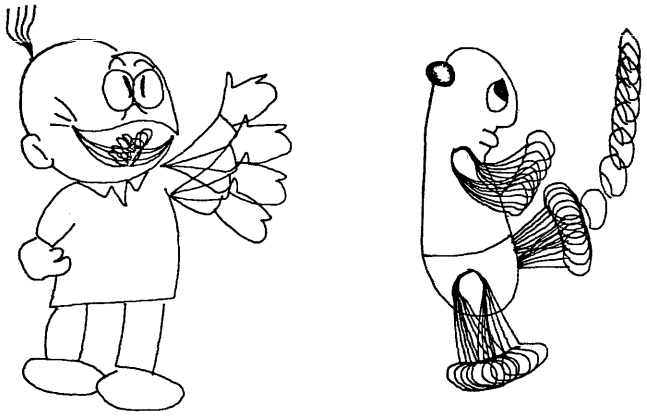


Figure 13 – Examples of animated cartoon

Results of application and their consideration

Animated cartoons were actually prepared on the basis of the methods described above. A typical example is shown in Figure 13. As is evident from the figure, the pictures can be moved about quite freely by this method, and the animated cartoons made in this manner were more or less satisfactory. In the future, when further improvements have been incorporated in the picture input device and the digital program, it is expected that this system will be quite satisfactory from the practical standpoint.

In the system described above, the pictorial patterns are read in as planar patterns, and new patterns are formed by specifying movements on the flat planar surface. However, animated cartoons are fundamentally projections on a plane surface of the movements of spatial objects.

In order to approach this problem, one may formulate, by the trial and error method, an equation corresponding to a clay model such as that shown in Figure 14, and the eyes and mouths can be produced by drawing frontal diagrams. Their x-y coordinates can be read, and assuming them to be present on this curved surface, one can calculate z from the above equation. If the spatial pattern is prepared in this way as an equation, one can draw a diagram of its projection on a flat surface as long as the center, the scale, and the direction of the designated points have been specified.

This is, however, merely a provisional method which can be used only for limited purposes.

It is found extremely advantageous to use hybrid techniques in order to meet the demand of faster curve displays with lower computer memory requirements. In our experiments, however, still a large amount of computing tasks were done in the digital computer, i.e. calculation of tangents. It was desired to develop an analog curve generator capable of generating a smooth curve given only successive point coordinates.

We have developed such a curve generator, which is now in use.

ACKNOWLEDGMENT

Many thanks are expressed to Researcher Fujii, who took charge of many sections of the actual work in the performance of this research project.

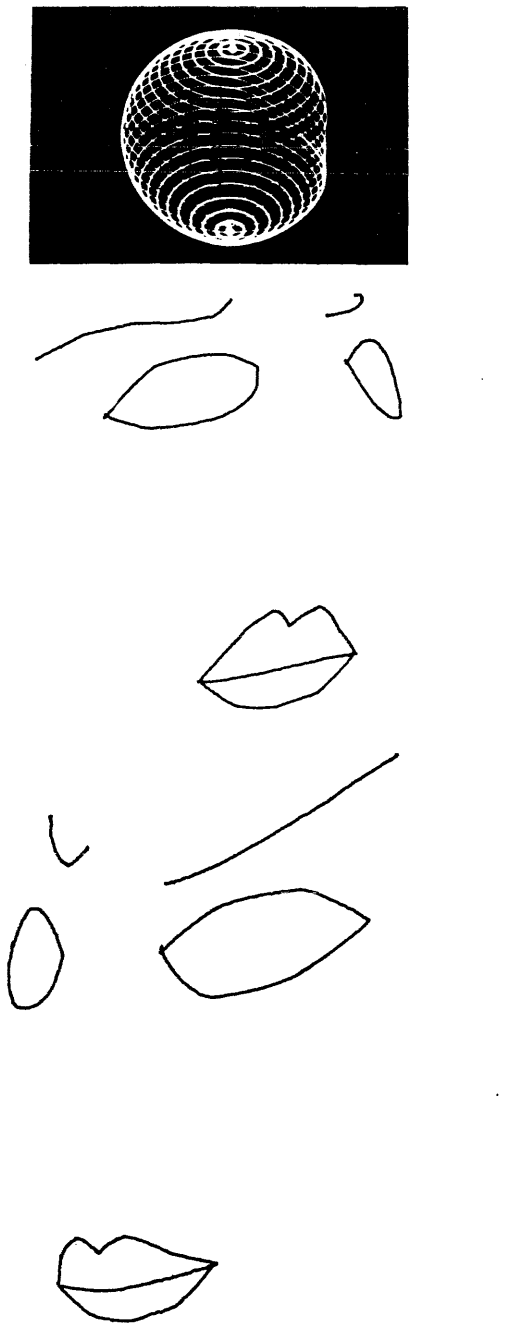


Figure 14—Three demensional display of an equation extracted from a clay model, and features drawn on its surface

Stochastic computing

by B. R. GAINES
Standard Telecommunication Laboratories
Harlow, Essex, United Kingdom

INTRODUCTION

The Stochastic Computer was developed as part of a program of research on the structure, realization and application of advanced automatic controllers in the form of Learning Machines.[1,2] Although algorithms for search, identification, policy-formation and the integration of these activities, could be established and tested by simulation on conventional digital computers, there was no hardware available which would make construction of the complex computing structure required in a Learning Machine feasible. The main problem was to design an active storage element in which the stored value was stable over long periods, could be varied by small increments, and whose output could act as a 'weight' multiplying other variables. Since large numbers of these elements would be required in any practical system it was also necessary that they be small and of low cost. Conventional analog integrators and multipliers do not fulfill requirements of stability and low cost, and unconventional elements such as electro-chemical stores and transfluxors are unreliable or require sophisticated external circuitry to make them usable.[3] Semiconductor integrated circuits have advantages in speed, stability, size and cost, and it was decided to design a computing element based on standard gates and flip-flops which would be amenable to large-scale integration.

A binary up/down counter has the properties of an incremental store, but requires many bits if the increments are too small and of variable size. If incrementing is made a stochastic process, however, 'fractional increments' may be effected in the stored count. Thus, if an increment of one-tenth of the value corresponding to the least significant bit is required, the counter may be incremented by unity with a probability of one-tenth—this is the basic principle of stochastic computing to represent analog quantities by the probability that an event will occur. This principle was first embodied in the ADDIE, a smoothing and storage device with stochastic input and output for realization of the STeLLA learning scheme,[1] and later extended to give rise to a family of computing elements capable of performing all the normal functions of an analog computer—this was called a Stochastic Computer.[4]

It is impossible within the scope of this paper to do more than describe briefly a few basic stochastic computing elements and configurations; Reference 4 discusses the theoretical basis of stochastic computing and describes some previous uses of random variables in data-processing, whilst Reference 5 describes the application of stochastic computing to process identification by means of gradient techniques, Bayes estimation and prediction, and Markov modelling.

Stochastic representation of numerical data

The stochastic computer is an incremental, or 'counting,' computer whose computations involve the interaction of unordered sequences of logic levels (rather than digital arithmetic between binary 'words'), and is in this respect similar to the Digital Differential Analyser,[6] Operational Hybrid Computer,[7] and Phase Computer.[8] In all these computers quantities are represented as binary words for purposes of storage, and as the proportion of ON logic levels in a clocked sequence (or frequency of pulses in the operational computers and asynchronous DDAs) for purposes of computation. In conventional computers, however, the sequences of logic levels are generated deterministically and are generally patterned or repetitious, whereas in the stochastic computer each logic level is generated by a statistically independent process; only the generating probability of this process is determined by the quantity to be represented.

This distinction is illustrated in Figure 1 which shows typical sequences in the various forms of computer: (a) is the output from a synchronous rate multiplier, or from the 'R register' of a DDA, corresponding to a store ¾ full—it will be noted that the ON and OFF logic levels are distributed as uniformly as possible; (b) is the output of a differential

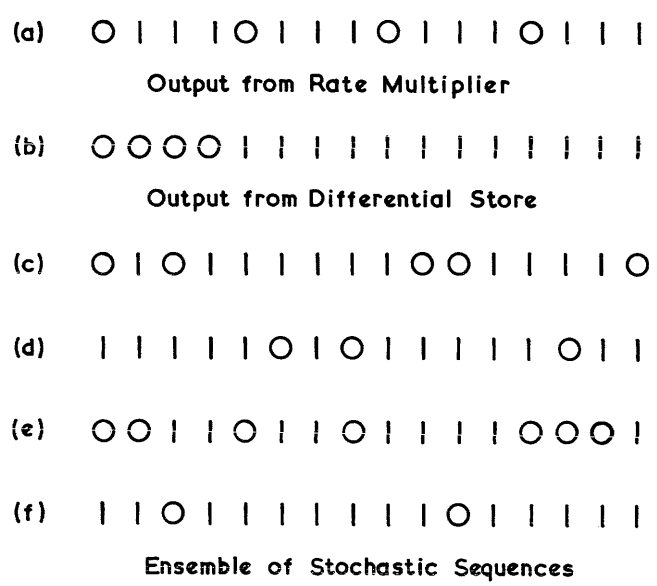


Figure 1—Typical computing sequences in various incremental computers

store in the Phase Computer—the OFF logic levels in a cycle all occur before the ON logic levels giving the synchronous equivalent of a mark/space modulated signal; (c) through (f) are examples of stochastic sequences with a generating probability of ¾—any sequence [including (a) and (b)] may occur, but the proportion of ON levels in a large sample of such sequences will have a binomial distribution with a mean of ¾.

Although a probability is a continuous variable capable of representing analog data without quantization error, this variable cannot be measured exactly and is subject to estimation variance. The effect of this variance on the efficiency of representation may be seen by comparing the number of levels required by various computers to carry analog data with a precision of one part in N:

- The analog computer requires one continuous level;
- The digital computer requires $\log_2 kN$ ordered binary levels;
- The DDA requires kN unordered binary levels; and
- The stochastic computer requires kN^2 unordered binary levels;

where $k > 1$ is a constant representing the effects of round-off error or variance, $k = 10$ say. The N^2 term for the stochastic computer arises because the expected error in estimating a generating probability decreases as the square-root of the length of sequence sampled.

Although this progression from 1: $\log_2 N$: N : N^2 shows the stochastic computer to be the least efficient in its representation of quantity, the lack of coherency or patterning in stochastic sequences enables simple hardware to be used for complex calculations with data represented in this way.

Stochastic computing

Although there are occasions when the [0, 1] range of probabilities may be used directly in computation, e.g. Bayes estimation and prediction,[5] it is usually necessary to map analog variables into this range both by scaling and shift of origin. Many mappings have been investigated, but the two considered in this paper are of particular interest because they give rise to computations similar to those of the conventional analog computer. These are:

(i) Single-line symmetric representation.—Given a quantity, E, in the range $-V \leq E \leq V$, represent it by a sequence with generating probability, p, such that:

$$p(ON) = \tfrac{1}{2}E/V + \tfrac{1}{2} \qquad (1)$$

so that maximum positive quantity is represented by a logic level always ON, maximum negative quantity by its being always OFF, and zero quantity by a random sequence with equal probability of being ON or OFF.

(ii) Dual-line symmetric representation.—The first representation suffers from the disadvantage that zero quantity is represented with maximum variance, and when values near zero must be distinguished it is better to use a two-line stochastic representation. Let the quantity, E, in the range $-V \leq E \leq V$, be represented by sequences on two lines, the UP and DOWN lines, such that:

$$p(UP = ON) = \begin{cases} E/V & \text{if } E > 0 \\ 0 & \text{if } E \leq 0 \end{cases} \qquad (2)$$

$$p(DOWN = ON) = \begin{cases} -E/V & \text{if } E < 0 \\ 0 & \text{if } E \geq 0 \end{cases} \qquad (3)$$

These equations give a minimum variance representation which is not necessarily maintained during a computation, and the most general form of the Dual-Line Symmetric Representation follows from the inverse equation:

$$E/V = p(UP = ON) - p(DOWN = ON) \qquad (4)$$

The next sections describe stochastic computing elements to perform the complete range of analog computing functions, inversion, multiplication, addition, integration, and so on, using synchronous logic elements acting on stochastic sequences.

Stochastic invertors, multipliers and isolators

To multiply a quantity in representation (ii) by −1 requires only the interchange of UP and DOWN lines. In representation (i) the logical invertor, whose output is the complement of its input, performs the

same function. Consider the relationship between the probability that its output will be ON, p', and the

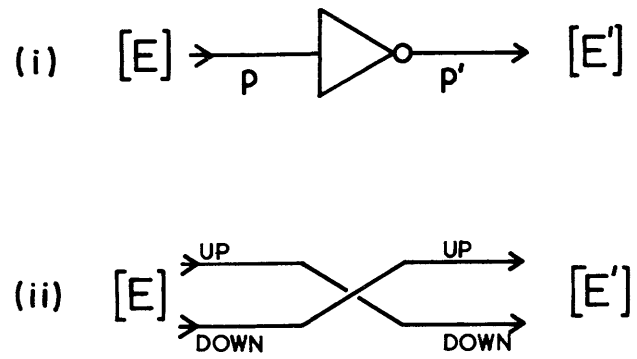


Figure 2—Stochastic invertors

probability that its input will be ON, p; that is

$$p' = 1-p \quad (5)$$

From equation (1), the relationship between these probabilities and the quantities they represent is:

$$p = \tfrac{1}{2}E/V + \tfrac{1}{2} \quad (6)$$

$$p' = \tfrac{1}{2}E'/V + \tfrac{1}{2} \quad (7)$$

hence

$$E' = -E \quad (8)$$

Multiplication of one quantity by another may be effected by an inverted exclusive-OR gate in representation (i), and by a pair of similar gates in representation (ii); realizations of these stochastic multipliers in NAND logic are shown in Figures 3(i) and 3(ii) respectively.

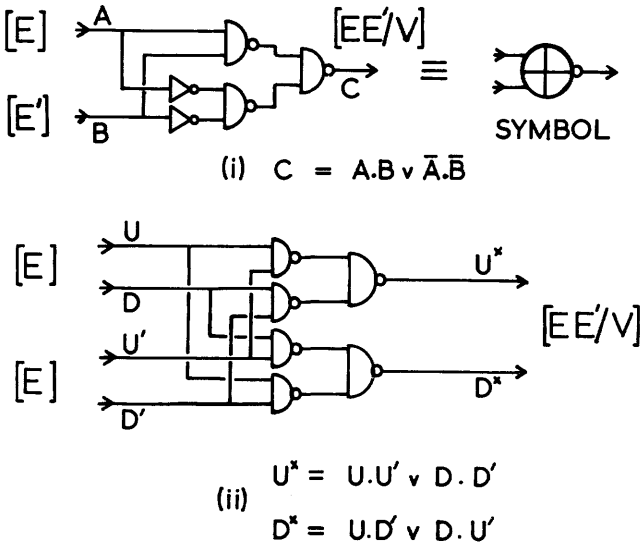


Figure 3—Stochastic multipliers

That multiplication does occur may be confirmed by examination of the relationship between input and output probabilities for the gates of Figure 3. For 3(i) we have:

$$p(C) = p(A)p(B) + (1-p(A))\,(1-p(B)) \quad (9)$$

and from equation (1):—

$$p(A) = \tfrac{1}{2}E/V + \tfrac{1}{2} \quad (10)$$

$$p(B) = \tfrac{1}{2}E'/V + \tfrac{1}{2} \quad (11)$$

so that:—

$$p(C) = \tfrac{1}{2}(EE'/V)/V + \tfrac{1}{2} \quad (12)$$

which is normalized multiplication of E by E'. A similar result is obtained for 3(ii) by substitution from equation (4) in the relationships:—

$$p(U^x) = p(U)p(U') + p(D)p(D') - p(U.U'.D.D.') \quad (13)$$

and:—

$$p(D^x) = p(U)p(D') + p(D)p(U') - p(U.U'.D.D') \quad (14)$$

An important phenomenon is illustrated by the use of a stochastic multiplier as a squarer. For example, it is not sufficient to short-circuit the inputs of the gate in Figure 3(i), for its output will then be always ON. This difficulty arises because we have assumed that the stochastic input sequences are statistically independent in obtaining equation (9) above. Fortunately an independent replication of a stochastic sequence (in fact a Bernoulli sequence) may be obtained by delaying it through one event, and Figure (4) illustrates how a squarer may be constructed using the multiplier of Figure 3(i) together with a flip-flop used as a delay; similarly the multiplier of Figure 3(ii) may be used as a squarer by feeding delayed replicas of U and D to U' and D' respectively. Flip-flops used in this way act as stochastic 'isolators', performing no computation but statistically isolating two cross-correlated lines.

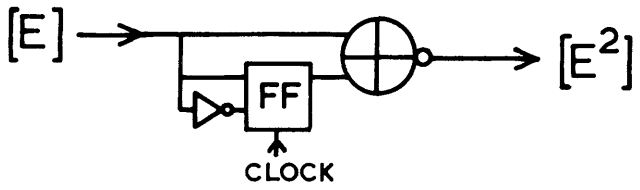


Figure 4—Stochastic squarer

Stochastic summers

Having seen how readily inversion and multiplication may be performed by simple gates, one is tempted to assume that similar gates may be used to perform addition. However this is not so, and stochastic logic elements must be introduced to sum the quantities represented by two stochastic sequences. For example, consider two stochastic sequences in representation (i), one representing maximum positive quantity and hence always ON, the other representing maximum negative quantity and hence always OFF. The sum of these quantities is zero, and this is represented by a stochastic sequence with equal probabilities of being ON or OFF. A probabilistic output cannot be obtained

from a deterministic gate with constant inputs, so that stochastic behaviour must be built into the summing gates of a stochastic computer.

Stochastic summers may be regarded as switches which, at a clock-pulse, randomly select one of the input lines and connect it to the output. The output line then represents the sum of the quantities represented by the input lines, weighted according to the probability that a particular input line will be selected. The random selection is performed by internally-generated stochastic sequences, obtained either by sampling flip-flops triggered by a high bandwidth noise source, or from a delayed sequences of a central pseudo-random shift-register; these sequences we call 'digital noise.'

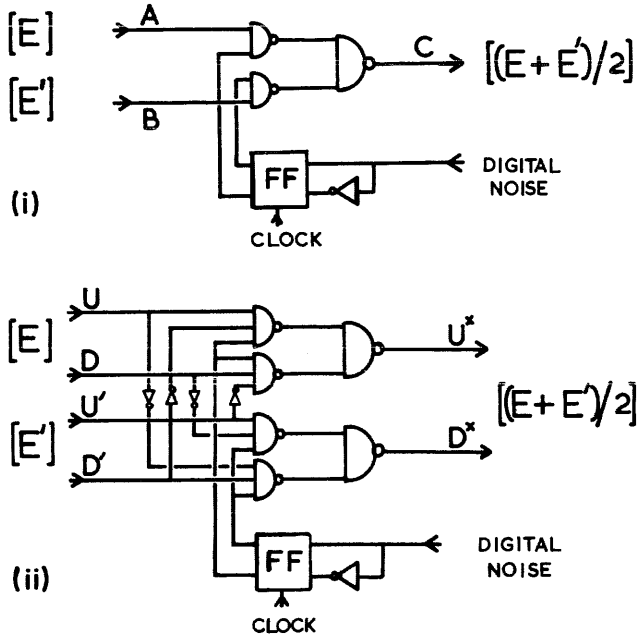


Figure 5—Stochastic summers

Two-input stochastic summers for quantities in representations (i) and (ii) are shown in Figures 5(i) and 5(ii) respectively; the cross-coupling between the inputs in Figure 5(ii) reduces the variance of the output. That addition does occur may be confirmed by examination of the relationship between input and output probabilities for the gates of Figure 5. For 5(i), assuming symmetrically distributed digital noise, we have:—

$$p(C) = \tfrac{1}{2}p(A) + \tfrac{1}{2}p(B) \qquad (15)$$

and hence from equations (10) and (11):—

$$p(C) = \tfrac{1}{2}(\tfrac{1}{2}(E + E'))/V + \tfrac{1}{2} \qquad (16)$$

which is normalized summation of E and E'. A similar result is obtained for 5(ii) by substitution from equation (4) in the relationships:—

$$p(U^x) = \tfrac{1}{2}p(U)\ (1-p(D')) + \tfrac{1}{2}p(U')\ (1-p(D)) \qquad (17)$$

$$p(D^x) = \tfrac{1}{2}p(D)\ (1-p(U')) + \tfrac{1}{2}p(D')\ (1-p(U)) \qquad (18)$$

Stochastic integrators, the ADDIE and interface

The basic integrator in a stochastic computer is a reversible counter—if this has N+1 states, then the value of the integral when it is its k'th state is:—

$$\int = (2k/N - 1)V \qquad (19)$$

If this quantity is to be used in further computations it must be made available as a stochastic sequence, and this may be generated by comparing the binary number in the counter with a uniformly distributed, digital random number (obtained from a central pseudo-random shift-register or a sampled cycling counter). In representation (i) the integrator output line is ON if the stored-count is greater than the random number. In representation (ii) the stored count is regarded as a number in twos-complement notation, whose magnitude is compared with the random number, and whose sign together with the result of this comparison determines whether the UP or DOWN output lines shall be ON.

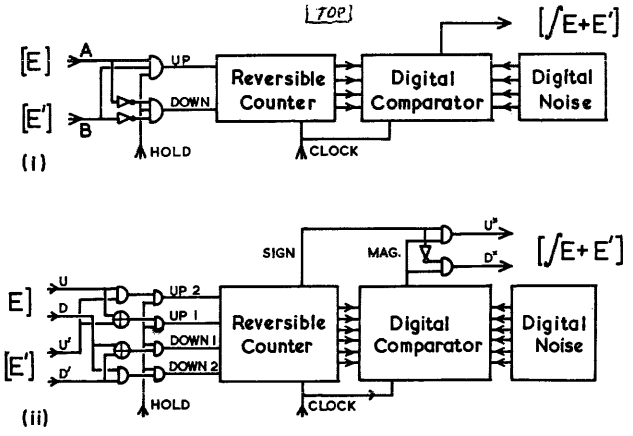


Figure 6—Stochastic integrators

Figure 6 shows two-input stochastic integrators for each representation with output lines as described above, and gating at the input of the counters to form the sum of the quantities represented by the input lines (stochastic summing is not required because the number of lines used to represent the sum is greater than the number of lines used to represent each input). A HOLD line at the input of the integrators determines whether they are in the integrate or hold modes, and

the normal integrator symbol is used for the overall device as shown in Figure 7.

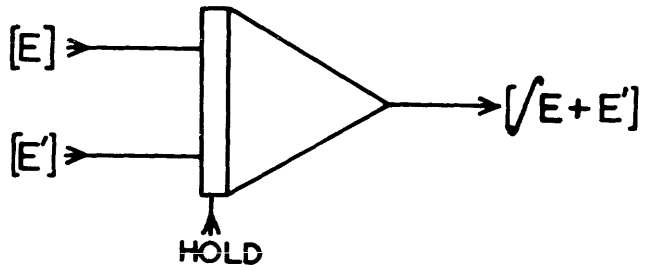


Figure 7—Integrator symbol

That integration does occur may be confirmed by examination of the expected increment, δ, in the counter at each clock-pulse. For 6(i):—

$$\delta = \frac{2V}{N}[p(A)p(B) - (1-p(A)\ (1-p(B))] \quad (20)$$

$$= (E + E')/N \quad (21)$$

by substitution from equations (10) and (11), which is equivalent to an integrator with a time-constant:

$$T = N/f \quad (22)$$

where f is the clock-frequency. A similar result may be obtained for 6(ii) by substituting from equation (4) in the relationship:—

$$\delta = \frac{2V}{N}[2p(U)p(U') + p(U)\ (1-p(U')) + p(U')\ (1-p(U)) - 2p(D)p(D') - p(D)\ (1-p(D')) - p(D')\ (1-p(D))] \quad (23)$$

$$= \frac{2V}{N}[p(U) - p(D) + p(U') - p(D')] \quad (24)$$

$$= 2(E + E')/N \quad (25)$$

which is equivalent to integration with a time constant:

$$T = N/2f \quad (26)$$

where f is the clock-frequency.

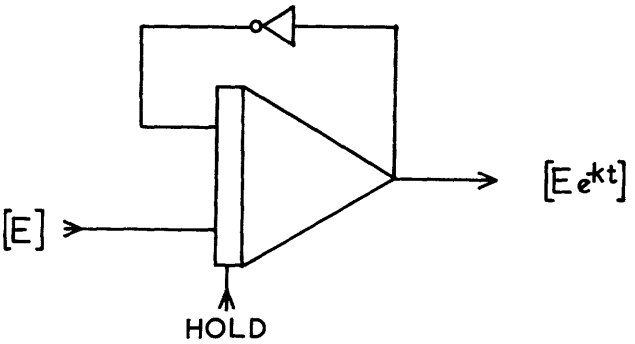


Figure 8—ADDIE

The integrator with unity feedback illustrated in Figure 8 is called an ADDIE, and performs the important function of exponentially averaging the quantity represented by its input. In terms of the quantity represented it may be regarded as a transfer function: 1/(s+1), and in terms of the stochastic sequences it may be shown that the fractional count in the store tends to an unbiased estimate of the probability that the input line will be ON at a clock-pulse (for the integrator of Figure 6(i) connected as in Figure 8). That is, for an N+1 state store in its k'th state:

$$\hat{p}(\text{INPUT} = \text{ON}) = k/N \quad (27)$$

with an estimation time of order N clock-pulses, and a final variance:

$$\sigma^2(\hat{p}) = p(1-p)/N \quad (28)$$

Thus any quantity represented linearly by a probability in the stochastic computer may be read out to any required accuracy by using an ADDIE with a sufficient number of states, but the more states the longer the time-constant of smoothing and the lower the bandwidth of the computer. Since the distribution of the count in the integrator is binomial, and hence approximately normal for large N, variables within the stochastic computer may be regarded as degraded by Gaussian noise whose power increases in proportion to the bandwidth required from the computer.

Integrators or ADDIEs form the natural output interface of the stochastic computer. Integrators with their HOLD lines OFF also form the input interface for digital or analog data, since binary numbers may be transferred directly into the counter to generate a stochastic output sequence, and analog quantities may be converted to binary form by comparison with a standard ramp generated by a cycling counter driving a digital/analog convertor. Similarly an integrator may be used to hold a constant and thus act as a 'potentiometer' if coupled to a multiplier. Arbitrary functional relationships may be realized by imposing a suitable nonlinear relationship between the stored count and the stochastic output; for example, an integrator whose output is ON when the count is equal to or above mid-value, and OFF when it is below mid-value [in representation (i)] approximates to a switching function.

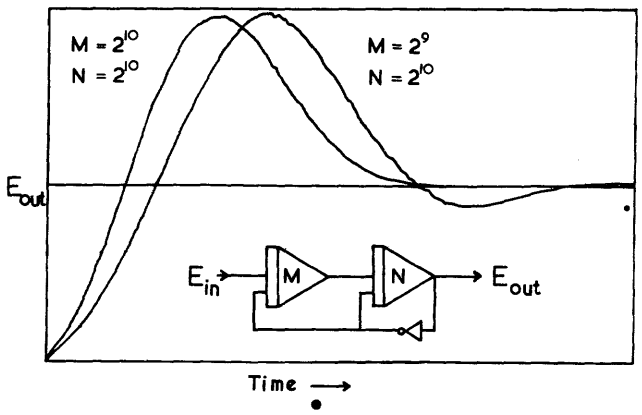


Figure 9—Stochastic second order transfer function—response to unit step in position and velocity

Higher-order smoothing than that of the ADDIE may be realized by connecting integrators in cascade with appropriate feedback loops. The inset of Figure 9 shows a stable second-order stochastic transfer-function using two stochastic integrators in representation (i). If the first integrator has M+1 states, and the second N+1, then the transformation realized is:—

$$\frac{MN}{f^2} E_{out} + \frac{M}{f} E_{out} + E_{out} = E_{in} \quad (29)$$

where f is the clock-frequency. So that the undamped natural frequency is:

$$f_n = \frac{f}{2\Pi(MN)^{1/2}} \quad (30)$$

and the damping ratio is:

$$\Sigma = \tfrac{1}{2}(M/N)^{1/2} \quad (31)$$

The responses to unit step in position and velocity for the two conditions: $M=2^{10}$, $N=2^{10}$ and $M=2^9$, $N=2^{10}$: are shown in Figure 9; these were obtained on the Mark I Stochastic Computer at STL.

Generation of stochastic sequences

The central problem in constructing a stochastic computer is the generation of many stochastic sequences (K for each integrator, where K is the number of flip-flops in the integrator counter), which are neither cross-correlated nor autocorrelated, and which have known, stable generating probabilities. This reduces to a requirement for a number of independent sequences each with a generating probability of ½, since any probability may be expressed as a fractional binary number and realized to any required accuracy by appropriate gating of a set of lines equally likely to be ON or OFF. For example, Figure 10 illustrates one technique for generating stochastic sequences with a generating probability of ½ by sampling flip-flops toggling rapidly from a noise source: the following NAND gates convert two of these sequences to one with a generating probability of ¾. This may be confirmed from the relationship:—

$$p(C) = (1-p(A)) + p(A)p(B) = \tfrac{3}{4} \quad (32)$$

The generation of digital noise as shown in Figure 10 is quite attractive since radio-active or photon-emitting sources may be coupled directly to semiconductor devices to form random pulse generators. It suffers from the disadvantage that very high toggling rates must be attained in FF_1 and FF_1' and sampled by very narrow strobes in FF_2 and FF_2', if a high clock-frequency is to be used.

The Mark I Stochastic Computer had six ten-bit integrators, each with their own internal digital noise source consisting of ten-bit counters cycling at a high clock-frequency. These counters were sampled at a very much lower and anharmonic clock-frequency to

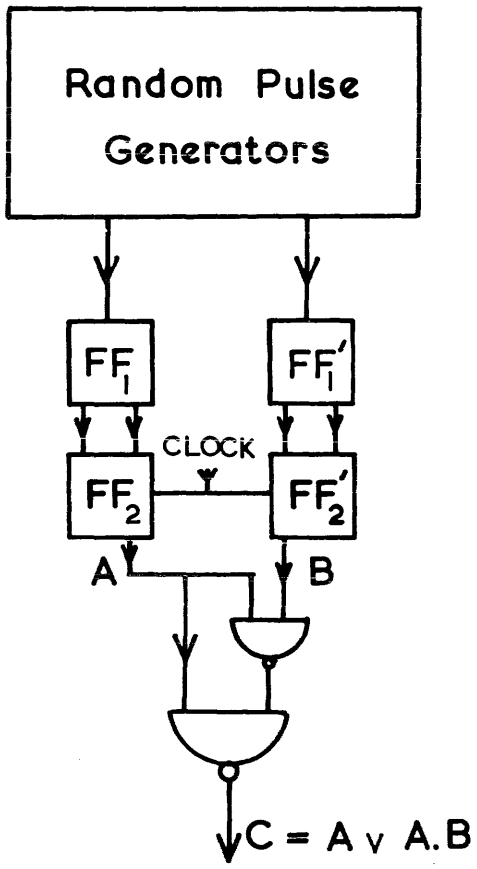


Figure 10—Generation of stochastic sequence (p=¾)

give an effectively random output. This was not a practical arrangement since the sampling frequency had to be so low (500 cs), that the overall bandwidth of the computer was only 0·1 cs ! ; as an experimental tool, however, it has enabled us to check out configurations, such as that of Figure 9, whose behaviour is difficult to determine theoretically.

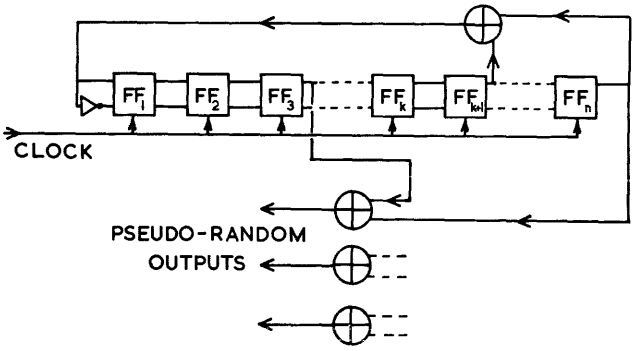


Figure 11—Central pseudo-random generator

The Mark II Stochastic Computer, at present under construction, uses entirely different techniques which have reduced both size and cost and increased the clock-frequency to 1Mcs. A single, central pseudo-random shift-register[9,16] is used to generate stochastic

sequences for all computing elements. Different sequences for each element are obtained by appropriate exclusive-OR gating of the shift-register outputs, giving delayed replicas of the sequence in the shift register itself. Such a generator is illustrated in Figure 11, and with 43 flip-flops it is capable of supplying 100 16-bit integrators for one hour without cross-duplication.

Serial arithmetic is used in the integrators of the Mark II computer so that the counter may be realized using shift registers and the comparators by far fewer gates. In this way a sixteen-bit stochastic integrator may now be fabricated from only six dual in-line packages. A clock-frequency of 16 Mcs in the shift-registers gives rise to a clock-frequency of 1 Mcs in the stochastic computer, and respectable bandwidths of 100 cs or so may now be attained.

Applications of stochastic computers

The Stochastic Computer was developed for problems arising in automatic control, and immediate applications are apparent mainly in the fields of adaptive control and adaptive filtering. Gradient techniques for process identification and on-line optimization are the simplest examples of powerful control methods which lack the hardware necessary for their full exploitation. Direct digital control using conventional computers is not practical with a small plant, and conventional analog computers are expensive because of the large numbers of multipliers required. Two-level or polarity-coincidence multiplication[10,11] has been suggested[12] as one means of realizing gradient techniques cheaply and reliably using digital integrated circuits; however a comparison of six techniques for multiplication in a steepest-descent computation has shown that, for the same convergence-time, polarity-coincidence and relay multiplication give much greater variance in parameter estimates than does the equivalent stochastic-computing configuration.[5] In this particular computation the stochastic multiplier may be regarded as a statistically-linearized[13,14] polarity-coincidence multiplier, and the addition of sawtooth dither to the input signals, which has been suggested as a means of effecting such linearization,[11,15,16] may be seen as a technique for obtaining a pseudo-stochastic sequence.

Maximum likelihood prediction based on Bayes inversion of conditional probabilities is the basis of many 'learning machines' for control and pattern-recognition, but the equations for estimation and prediction are difficult to realize with conventional computing elements, and a stored-program digital computer has been required for experiments with this technique. Using a three-input variation of the ADDIE, however, the estimation of normalized likelihood ratios, and prediction based on them, become very simple operations requiring little hardware.[5]

The theoretical basis for the economy in hardware offered by stochastic computing lies in a theorem of Rabin[17] and Paz,[18] to the effect that a stochastic (or probabilistic) automaton is equivalent to a deterministic automaton which generally has more states. An immediate practical example of this phenomenon may be found in adaptive threshold logic as used in pattern-classifiers such as the Perceptron[19] or Adaline.[20] An adaptive threshold logic element with discrete weights will not necessarily converge under the Novikoff conditions,[21] even though the weights can take values giving linear separation, whilst the equivalent stochastic element may be shown to converge under the same conditions.[5]

A pictorial explanation of this difference is that the direction of steepest descent followed in adaptive threshold logic with continuous weights cannot be taken if the weights are discrete, and there are then several directions of 'almost steepest descent.' Deterministic logic has to 'chose' one of these directions and, if it is the 'wrong' one, may get into a cycle of wrong decisions, whereas stochastic logic has a probability of taking any of the possible directions of descent and is bound to take the right one eventually.

CONCLUSION

The main performance measure of a computer are size and range of possible problems, speed and accuracy of solution, and physical size, reliability and cost of the computer. There are strong interactions between these measures and it is unlikely that any one form of computer will ever be optimal on all counts. The identification and simulation of complex processes, and the realization of multi-variable control systems, requires large numbers of computing elements such as multipliers, summers and integrators, working simultaneously and costing little. However these elements do not have to compute a solution quickly or accurately, for a bandwidth of 10 cs and an overall accuracy of 1% is adequate in the simulation of economic and chemical processes, and in control systems where feedback is operative a computational accuracy of 10% may be ample. In these situations it is advantageous to trade the accuracy of the digital computer and the speed of the analog computer, for the economy of the stochastic computer.

ACKNOWLEDGMENTS

My thanks are due to Dr. J. H. Andreae (now at University of Canterbury, Christchurch, N.Z.) for long discussions on learning machines and computers, and for his criticism of this manuscrapt; to Mr. P. L. Joyce

for constructing the first Stochastic Computer and obtaining the results of Figure 9, and to S.T.L. for permission to publish this paper.

REFERENCES

1 J. H. ANDREAE *Learning machines* in Encyclopaedia of Information, Linguistics and Control (Pergamon Press, to be published)

2 B. R. GAINES and J. H. ANDREAE *A learning machine in the context of the general control problem* Proceedings of the 3rd Congress of IFAC 1966

3 G. NAGY *A survey of analog memory devices* IEEE Trans. Electron. Comp. 12 388 1963

4 B. R. GAINES *Stochastic computers* in Encyclopaedia of Information, Linguistics and Control Pergamon Press, to be published

5 B. R. GAINES *Techniques of identification with the stochastic computer* Proc. IFAC Symp. Problems of Identification 1967

6 F. V. MAYOROV and Y. CHU *Digital differential analysers* Iliffe Books, London 1964

7 H. SCHMID *"An operational hybrid computing system* IEEE Trans. Electron Comp. 12 715 1963

8 B. R. GAINES and P. L. JOYCE *Phase computers* 5th AICA Congress 1967

9 W. W. PETERSON *Error correcting codes* MIT Press & Wiley, New York 1961

10 C. L. BECKER and J. V. WAIT *Two-level correlation on an analog computer* IRE Trans. Electron Comp. 10 752 1961

11 B. P. TH. VELTMAN and A. van den BOS *The applicability of the relay correlator and the polarity coincidence correlator in automatic control* Proc. 2nd Congress IFAC 1963

12 P. EYKHOFF, P. M. van der GRINTEN, H. KWAKERNAAK, B. P. TH. VELTMAN *Systems modelling and identification* Survey Paper 3rd Congress of IFAC 1966

13 O. I. ELGERD *High frequency signal injection: a means of changing the transfer characteristics of nonlinear elements* WESCON 1962

14 A. A. PERVOZANSKII *Random processes in nonlinear control* Academic Press, New York 1965

15 P. JESPERS, P. T. CHU, and A. FETTWEIS *A new method to compute correlation functions* Proc. Int. Symp. Inf. Theo. 1962

16 G. A. KORN *Random-process simulation and measurement* McGraw Hill, Inc. 1966

17 M. O. RABIN *Probabilistic automata* Inf. & Contr. 6 230 1963

18 A. PAZ *Some aspects of probabilistic automata* Inf. & Contr. 9 26 1966

19 F. ROSENBLATT *A model for experimental storage in neural networks* in Computer and Information Sciences Spartan Books, Washington, D.C. 1964

20 B. WIDROW and F. W. SMITH *Pattern-recognizing control systems* in Computer and Information Sciences Spartan Books, Washington, D.C. 1964

21 A. NOVIKOFF *Convergence proofs for perceptrons* in Mathematical Theory of Automata Polytechnic Press, Brooklyn & Wiley Interscience 1963

The people problem: computers can help

by E. R. KELLER, 2nd and S. D. BEDROSIAN
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

Each day tens of millions of people parade in and out of our academic and industrial facilities in search of their daily bread. Of these, many find their work challenging and stimulating. Countless others, however, are reduced to vestiges of human spirit by the frustrations of their environment, boredom with lack of responsibility, or misdirection of their talents. Consequently, they approach their assignments with something less than a maximum of enthusiasm.

Meanwhile each day thousands of holes in personnel pegboards go unfilled. Positions requiring skill at all levels remain unclaimed because of inadequate or unavailable descriptions of qualified workers. Personnel Departments charged with the responsibility of recruiting, retaining, and developing effective human support for their respective organizations are often hopelessly frustrated by their inability to bring together people and jobs. From the standpoint of computer implementation we may interpret this as the problem of mapping a set of tasks into a universe of talents.

A natural recourse, for people so inclined, is to attempt to manage this massive matching operation through computer mechanization. As evidenced by pertinent articles in such magazines as Business Week,[1,2,3] American Education[4] and Dun's Review,[5] the people problem is far from being ignored. Although the mechanization of personnel information is by no means a novel concept, the mass storage of comprenehsive analyses of individuals is in its infancy. Of the many personnel data systems now in existence, those of Esso[1] and IBM[2] seem to be the most comprehensive and coordinated. From a cursory inspection, however, even those do not appear to treat the many details necessary to exploit fully the employee's desires and capabilities for future work, especially if unrelated to his present (or past) assignment. Through the use of existing techniques in mechanized inference-making[6] and by judicious application of on-line storage and retrieval systems [6,7,8,9] it should be possible to develop and maintain a work force consisting primarily of interested workers. One can envision for instance filling a vacated position with someone from a relatively unrelated position by exploiting his competence in a hobby.

We are concerned more with the general problem of storage and retrieval systems than with a particular application of them. In fact, the system described in this paper is directed toward flexibility of application rather than toward implementation of a specific file type. In order to demonstrate the utility of the system, data for use in resolving the people problem are being collected, stored and applied at the University of Pennsylvania.

System organization

The filing system

The communication language and the system of interpretive processors used for maintaining the information files and for manipulating the information during retrieval and output are named, collectively, GENERAL STORE. The overall system layout is shown in Figure 1. Although an input item to GENERAL STORE may be constrained in its form or its relation to other items in storage at the discretion of the user, GENERAL STORE is indiscriminate as to the nature of information stored in its files. For instance in the pilot model, a file of mechanical and electrical descriptions of electronic assemblies for use in system design and test is stored along with personnel information.[10] In addition the user is permitted a maximum of flexibility in describing procedures for controlling file maintenance, interfile retrieval, and manipulation of data for output.

We turn now to the specific objectives in the conceptual design of GENERAL STORE which have been implemented in greater or less degree. In gen-

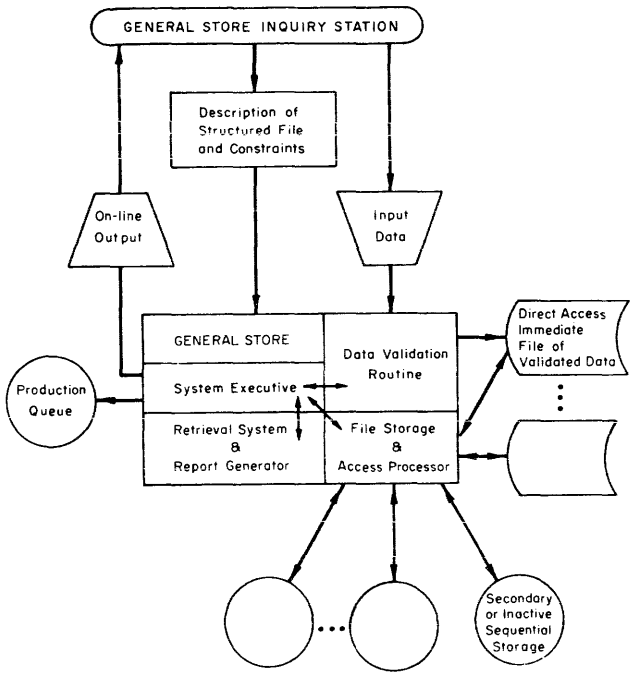


Figure 1—General store system diagram

Outline Item	Description
1	Document Name
2	(Authorship)
2.1	(Author 1)
2.1.1	Name
2.1.2	Address
2.1.3	Affiliation
2.2	(Author 2)
2.2.1	Name
2.2.2	Address
2.2.3	Affiliation
3	(Publisher)
3.1	Publisher's Name
3.2	Publisher's Address

Figure 2—General store fixed file description

eral, however, the following discussion focuses on our goals rather than accomplishments. In fact, the GENERAL STORE concept could be implemented, with some reservations, as an extension of many existing on-line systems of which the Problem Solving Facility at the University of Pennsylvania[7] is an example.

List structured data files

Just as the technique of preparing an outline is a familiar and powerful organizational tool for nearly everyone who has set pen to paper, the structuring of data records is becoming a commonplace solution to the problems of storing variable length, multiple entry, or historical information. This development is due in the main to LISP,[11] IPL-V,[12] and certain other existing list processing languages[13,14,15,16] through which manipulation of complicated data structures is simplified.

In concept, each data file in GENERAL STORE has associated with it an ordinary outline describing the contents of a record in that file. By means of this outline both the user and the machine can establish a useful means of communicating information about the file so defined.

In Figure 2 is shown an example of an elementary bibliographic file as the description might be given to GENERAL STORE. The numbers represent a means by which the user may reference each line of the outline, and names are symbols by which the user may refer to each line. Some lines are contained in parentheses to indicate that these lines never actually contain data but are used only as classifications for, or pointers to, the items subordinate to them in the structure. These lines represent intermediate entries in the end-point code for each data line. This figure contains locations in the structure for information about two authors. As shown in Figure 3, the concept of multiple authorship can be indicated more concisely by replacing the author number with a letter. This practice permits the letter specified to assume any integral value. In fact, the number of items at any level may be specified as a variable.

Data may be represented for input to GENERAL STORE in at least two ways. First, it is possible to specify an item for entry by stating a reference, either as a number or name from its outline description, followed by the pertinent information. In addition, GENERAL STORE employs a flexible system which permits specification of the line name by an abbreviation of the user's choosing. The second method involves writing the input information in a more conventional list notation where each sublist is enclosed in parentheses. Examples of these alternatives are shown in Figure 4.

Relocation of programs and data in available storage

Since GENERAL STORE is intended to be implemented on a computer having limited storage

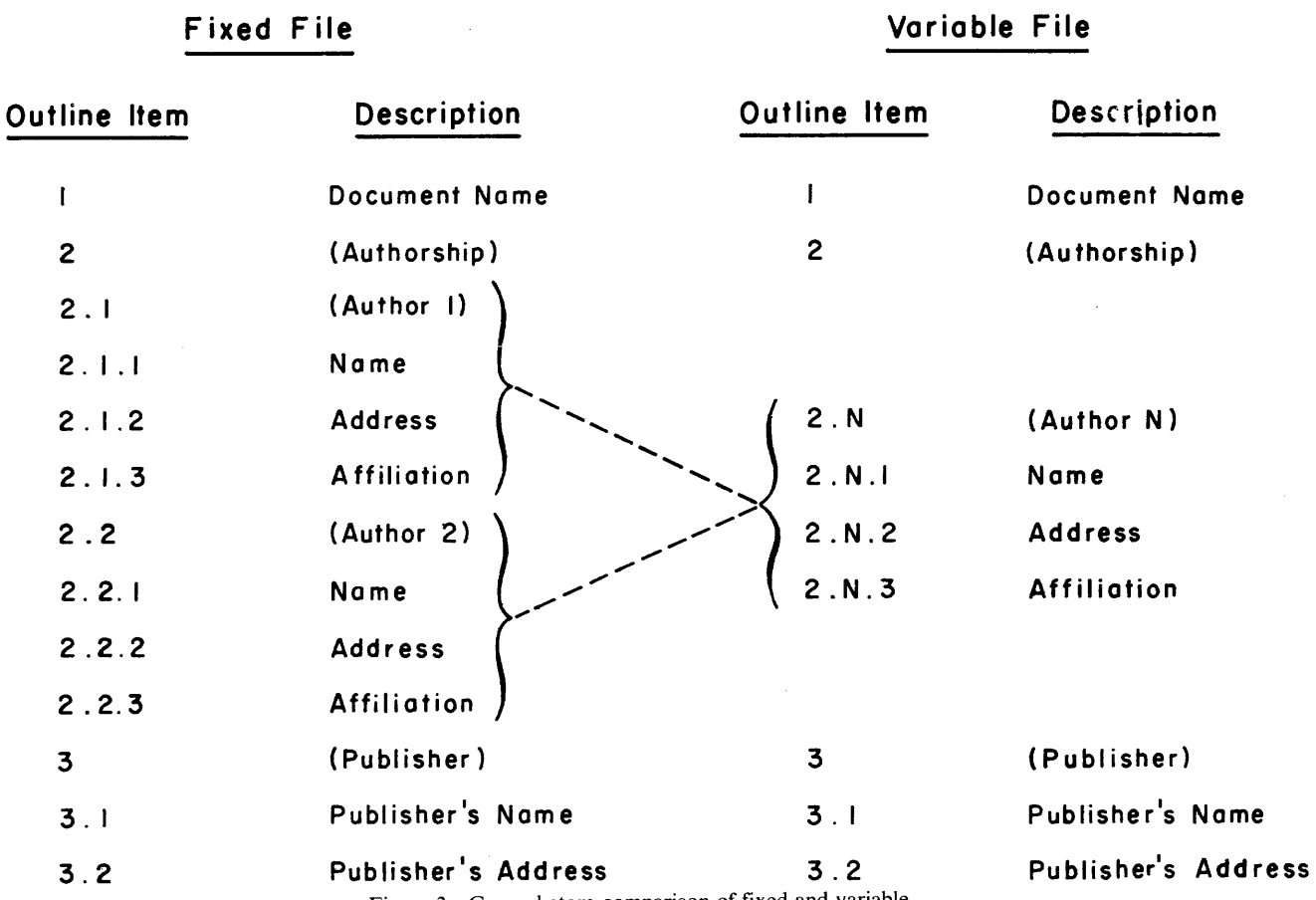


Figure 3—General store comparison of fixed and variable file descriptions

capacity in any one device, it has been necessary to design the system in such a way that only active programs and data are stored in the fastest memory devices.[17] The user, by specifying to which files he will need access, causes GENERAL STORE to arrange for as many of these files as possible to be loaded into direct-access devices. Files not so specified will still be available during processing, but may entail longer access times.

Flexible arithmetic and string manipulating functions

The user may in some cases request standard output formats and may give specific information for input. In this system he has been provided with a report-generating facility and the ability to describe procedures to be followed in selecting records for output or in preparing a record for output. By making available to the user a set of list-processing functions in a language understood by the GENERAL STORE Interpreter, or included as a program in the GENERAL STORE data files, he can perform extensive modification of any item. Such modification may be made either in terms of other items in the file or on the basis of information introduced at the time of modification.

User-defined constraints on input data

In preparing an outline to describe a file, the user may include certain restrictions to be placed on the data that will fill each outline item. In this way, a considerable amount of checking can be done by the processor to help insure the validity of input data. For example, the constraints may simply require that the data contain only numeric characters, or that the input field be a particular fixed length. Again, constraints specifying a relationship among parameters in this or other files may be stipulated. This procedure is demonstrated by the personnel file described in Figure 5. Shown with the description are a few examples of constraints which may be imposed on input data. Of particular interest is the restriction of total salary to a value greater than or equal to the sum of all individual project salaries.

Interfile reference

Just as in specifying constraints, the boundaries of a single file may be crossed at will when specifying criteria for retrieval. In fact, it may be convenient to think of each data file as a sublist of items in the master file structure. It is exactly in this way that programs and data fit into the GENERAL STORE concept.

(2.1.1 = JONES, BT)

(2.1.2= 3120 WALNUT ST.-PHILA., PA.)

(2.1.3 = UNIV. OF PENNA.)

REFERENCE BY NUMBER

OR

(AUTHOR 1/NAME = JONES, BT)

(AUTHOR 1/ADDRESS = 3120 WALNUT ST.-PHILA., PA.)

(AUTHOR 1/AFFILIATION = UNIV. OF PENNA.)

REFERENCE BY NAME

OR

2 = ((JONES, BT)(3120 WALNUT ST.)(UNIV. OF PENNA.))

CONVENTIONAL LIST ENTRY

Figure 4—General store input forms

Outline Item	Description	Constraint
1	Employee Number	FORM ≤ XXXXX
2	Name	
3	(Address)	
3.N	(Address N)	
3.N.1	Date first occupied	
3.N.2	Address	
4	Phone	
5	(Earnings)	
5.1	Total Salary	FORM ≤ XXXXX.XX MAG ≥ $\sum_m$ (5.M.2)
5.M	(Project M)	
5.M.1	Fund Number	
5.M.2	Project M Salary	FORM = XXXXX.XX

Figure 5—Possible constraints on input data

Complete operating instructions

At any point a user may request guidance from the system as to what alternative steps are available to him, or how to proceed in accomplishing a specific goal. It should be possible, given the ability to gain access to the system, for an uninitiated user to bootstrap his way into useful, and possibly even optimal, operation of GENERAL STORE. For example, a prospective user need only know that GENERAL STORE contains information pertinent to his problem and that he can at any time request aid from the system by pressing an interrupt key. Following an affirmative response to the initial system-generated question:

DO YOU NEED HELP? TYPE (YES) OR (NO),

a dialogue ensues in which the user may be informed of the different types of file available and the data contained in each file. If a useful file is disclosed, a discussion may be requested concerning the elementary techniques for retrieving items from specific records. Logical statements are then written which define retrieval constraints on the items to be observed and on other items in the same record. As the user gains facility, he is introduced to techniques involving more complex functions of the form and magnitude of the data fields. If he wishes to spend the computer time and money, the user can eventually learn from the system itself virtually all of the operating details of GENERAL STORE.

Optional queueing of a job for deferred production

During any retrieval procedure, the approximate number of records which will result is noted by GENERAL STORE and may be requested in advance by the user. If either this number or the time estimate for completion of the job appears to be too large, the user has two options. First, he may reduce the scope of his request and reinitiate searching, or second, he may place the procedure and data in a production queue with a specified priority. Jobs in the queue will be performed in order of priority when the system is not involved with real time operation.

File protection

GENERAL STORE allows the user to modify the contents of a file not only by introducing data directly from the outside, but also by writing a procedure which will operate on items within the file as well as on external data. As mentioned earlier, it is often the case that constraints specified by the user will be adequate to check modification procedures. However, when it is felt that the constraints are not adequate to ensure proper updating, the user may request a display of the files as they would appear after modification but without actually introducing a permanent change. As a further hedge against unintentional destructive modification of its files, GENERAL STORE maintains a complete historical record of the transactions it has processed. This record may be used to recreate the file structure if necessary.

Delayed completion of on-line procedure

It is often the case that a user has not allocated enough on-line time to complete a particular procedure. By using the appropriate features of GENERAL STORE, the user can store his procedure and the intermediate results in such a way that they can be subsequently recalled. The resultant effect is one of no apparent interruption in processing.

Storage and modification of procedures

Although some procedures may be used only once and discarded when the results have been obtained,

it is more likely that others will be used repeatedly. This eventuality is accommodated by storing these complicated procedures with variations as required for execution on different sets of data.

A filing system having many of the characteristics of GENERAL STORE is being used to store personnel information on research workers at the Univeristy of Pennsylvania. By storing job descriptions, historical analyses, and the individual's own interpretation of his capabilities and his interests, it is anticipated that Personnel Directors will have a new tool with which to move closer to their goal of successfully matching jobs and people. Preliminary to full scale implementation of GENERAL STORE on an IBM System/360 computer, pilot runs are being made using programs developed for the IBM 1401 and IBM 7040.

Data collection

Collection of original and up-date information is by far the most time consuming, unreliable, and frustrating of all the tasks involved in operating the personnel file. This feature is by no means unique to the approach under discussion. Its solution relies heavily on the proper approach not only to each individual, but also to the managers of the institution or department for which he works. Through frequent, but unobtrusive, contact with the individuals, and by building an attractive, responsible, and worthwhile image of the system, it is hoped that the problems generally encountered in massive personnel data collection can be minimized.

The personnel file concept described herein is completely in step with the times. Never before has the people problem centered so specifically on getting the right person in the right job. Technological progress accompanied by social unrest has introduced new pressures on individuals and on employers. One has only to read current help-wanted advertising in any newspaper to see evidence of the present magnitude of a perennial problem which has been complicated by the introduction of new requirements based on new skills stemming from new technologies. One advertisement which appeared in the New York World Journal Tribune for October 20, 1966, was presented essentially in the form of a structured list with eight first-level descriptors, 22 second-level and more than 195 third-level descriptors. We submit that without competent assistance neither an individual nor his prospective employer can respond effectively to an advertisement written to this degree of specificity. In our judgment, the most promising approach to the "People Problem" at this time lies in full and intelligent cooperation between computers and people.

ACKNOWLEDGMENTS

The research on which this paper is based was performed at the Institute for Cooperative Research of the University of Pennsylvania. It was supported in part by the U. S. Navy under Contract NObsr-95215. The authors wich to acknowledge with thanks the stimulating discussions with Mr. A. M. Hadley and also critical evaluation of the manuscript for this paper.

REFERENCES

1 Describing Men to Machines Business Week 113 4 June 1966

2 *Computers move up in personnel ranks* Business Week 118 23 Oct 1965

3 *Picking top men* Business Week 97 6 March 1965

4 W O REED *Data link* American Education 31 1 June 1965

5 *Office services* Dun's Review 86 part 2 166 Sept 1965

6 M E MARON R LEVIEN *Relational data file: A tool for mechanized inference execution and data retrieval* 3rd National Colloquium on Information Retrieval University of Pennsylvania May 1966

7 N S PRYWES *A problem solving facility* Moore School of Electrical Engineering 20 July 1965

8 T OLLE *INFOL: A generalized language for information storage and retrieval application* Control Data Corp April 1966

9 *Generalized information system* IBM Application Program White Plains New York 1965

10 A M HADLEY et al *Project monidan final report on contract NObsr-95215 Dept of Navy BuShips* The Institute for Cooperative Research University of Pennsylvania 30 June 1966

11 J McCARTHY et al *LISP 1.5 programmer's manual* MIT Press Cambridge Mass 1962

12 NEWELL ALLEN *Information processing language V* Prentice-Hall Englewood Cliffs N J 1961

13 G LOEV *SLIP list processor* University of Pennsylvania Computer Center Philadelphia Pa April 1966

14 K C KNOWLTON *A programmer's description of L^6* Communications of the ACM 9 8 616 1966

15 *COMIT programmer's reference manual* Research Lab of Electronics MIT Cambridge Mass 1961

16 D J FARBER R E GRISWOLD I P POLONSKY *SNOBOL: A string manipulation language* Journal ACM 11 21 Jan 1964

17 W C McGEE On Dynamic Program Relocation IBM Systems J 4 3 184 1965

File handling at Cambridge University

by D. W. BARRON, A. G. FRASER, D. F. HARTLEY,
B. LANDY and R. M. NEEDHAM
The University Mathematical Laboratory
Cambridge, England

INTRODUCTION

The file handling facility now in use on the Titan* computer at the University Mathematical Laboratory, Cambridge provides storage for data over indefinitely long periods of time and yet contrives to make that data available on demand. An 8 million word disc is augmented by the use of magnetic tape. The organization of this available space is entirely a system responsibility and the user can normally expect data to be transferred into his immediate access store area on demand.

The filing system is part of a multi-programming system which has the ability to hold several user programs simultaneously in the core store, and provides on-line facilities for interacting with these programs. Jobs are initiated either off-line for normal job-shop working, or through console keyboards for interactive on-line working. Several of the basic concepts of the Titan system have been derived from the earlier supervisor written for the Atlas 1.[1] In particular, the concept of a storage "well" which is used to buffer all data passing to and from the "slow peripherals" was an essential part of the Atlas system. This well has been implemented at Cambridge as an integral part of the more versatile file handling mechanism now described. The concept of a standard internal code with interpretation for every peripheral is also carried over from the earlier work.

The file handling proposals for the Multics system[3] had an important influence on the design as did the experience gained during an earlier attempt at implementing a permanent storage facility.[2] The final choice is a compromise between a number of conflicting requirements. Some of these are listed below. This choice was not made easier by the realization that it would have a profound influence on the future pattern of machine use.

*Titan was developed from the prototype ICT Atlas 2

On designing a filing system

Paramount amongst the design requirements for a filing system is the need for confidence on the part of the user. To obtain this the protection given against the effects of system failure must be demonstrably adequate. The user must also be able to determine the extent of the protection provided against misuse of his files either by his own error or by the activities of others. On the other hand, the protection system must not be so designed that it absorbs an unwarranted amount of the available space or time, and the normal method of securing file access must not be too cumbersome. The natural patterns of file use (including group working) should not be distorted by system controls.

Protection against system failure implies some safeguard against the misplaced activities of the operating staff. Not only is it necessary to check operator actions but the system design should be such that non-standard activities are unnecessary and unreliable short cuts are eliminated. The latter may be expected to arise when the normal mechanism for recovery from system failure itself fails, and also if the recovery mechanism is protracted or too expensive in terms of lost computing activity.

Poorly designed controls and inadequate flexibility can give rise to unexpected patterns of activity. Not all of these will be foreseen, but the design must be checked against such variations as can be foreseen. For example, the system of priorities used for space allocation should not be so designed that it can be negated by a simple change in the user's pattern of behaviour.

The demand for space at Cambridge exceeds the supply of space on disc. Magnetic tapes are used to provide the necessary extra space and housekeeping operations are therefore required to regulate disc loading. The file store must, for operational purposes, appear to be uniform and some care is required to keep disc and tape transfer to a minimum. The limited

facilities provided by the hardware should not be totally concealed, however; some feedback which encourages the user to work along lines in keeping with the physical limitations of the system will prevent the extravagance which arises when a user is not aware of the fact that he is overloading the system.

Documents

The Titan system provides storage and handling facilities for documents, where a document is defined as any ordered string of data that is available to a user program, but is stored outside the area administered by that program. Access to a document is provided by system calls and the user program may use these to read, alter or create documented information.

Data that are input through the slow peripherals (paper tape and punched cards) are assembled into individual documents by the system. Each document is identified by a title that is supplied by the user, and is local to the job to which it belongs. Similarly, a user program may create documents which may be assigned individually to particular types of slow peripherals (line printer, paper-tape punch or plotter). The input and output processes themselves are time-shared supervisor activities that operate quite separately from the associated user program execution.

Jobs

The word *job* is used here to identify the connected series of activities initiated by one set of input documents, and one distinct activity is called a *phase*.

Each phase of a multi-phase job operates on some or all of the documented data from that job and may at the same time create new documents. The documents left by one phase can be used by the next and represent the major connection between the separate phases of one job. A typical example of a multi-phase job is the sequence: edit, compile, assemble and run.

The word job also identifies the series of activities initiated from an on-line console during the time that the user of the console is logged in. In this case it is most likely that the activity will be multi-phase.

Local and global data

The data flow to and from peripherals and the flow of data between phases is an organizational consideration which is not necessarily the concern of the user program. The commands which control the data flow and initiate each phase, resemble a simple programming language in which documents are the data elements.

For many command sequences it is necessary to recognize a number of intermediate documents which exist only to interface between distinct activities. Intermediate documents are also brought into existence to communicate with slow peripherals. These documents are the local data for the job concerned and for convenience their names have a scope which is local to the job; thus no confusion arises if two identical jobs are timeshared.

Global documents are required whenever data from one job are held over for subsequent processing by another, and for this purpose a document can be given a title that is recognized in a global context. Such documents are called *files*.

There are a number of important differences between local and global documents:

(a) The scope of the name.
(b) The life—local documents cease to be of value after the last phase of a job has been completed.
(c) Activity—a local document usually has a high rate of activity throughout its short life.
(d) Backup—by virtue of its short life and high activity a local document will usually be guaranteed high cost space. An infrequently used global document could be relegated to lower cost and less immediately available space.
(e) Protection—local documents are protected by the narrow scope of the document name. Global documents require a separate mechanism for protection against accidental or malicious misuse.
(f) Recovery—after a system failure the cost of recovering a lost local document will usually be no more than the cost of repeating one job. Global documents, which can be expected to be much greater in total volume, are not so easily recovered.

In view of these differences, two distinct software packages are used to provide the supporting facilities for the document handling system. One is concerned with local documents and the other is responsible for the files.

The facilities for processing a document are quite independent of the two support packages and the method of processing makes no assumptions about how a document is classified. Two types of processing are provided:

(a) Character string processing—The document is assumed to contain variable length records of characters in a standard format. Appropriate system calls are provided for character at a time and line at a time transfer.
(b) Literal processing—No format assumptions are made and the data is transferred in blocks of 512 48-bit words.

Disc space control

One system of space allocation is used for all documents, and all disc transfers are supervisor controlled. Files are also held on magnetic tape (see below) and space allocation here is the responsibility of the File Support package.

Document storage space on the disc is allocated in units of 1 block (512 words), and this is controlled by a *map* of the disc space. Each block of disc space is represented by a corresponding entry in the map, and a document is represented by linking together the appropriate entries. Free space is also linked through the map. It is important to note that this linking is not part of the data space, and the map is stored separately. This enables the system to locate the position of any block of a document without access to the document itself: for example, a document can be deleted from the disc by a simple rechaining operation within the map.

Parts of the map are held in the core store as and when space requirements permit, but they are dumped on to the disc when it is necessary to ease the demand for core space, and also periodically so as to permit recovery of documents in the event of system failure.

The remaining sections of this paper describe the various facilities and support packages of the filing system; further details of local document organization are dealt with in a further paper.

File identification and access

Any prospective user of the computer must gain access by quoting his initials and project number. This combination, known as his title, is required for off-line as well as on-line working, and is checked before any computing is permitted.

The title of the file owner is the first of three alphanumeric components which identify a file. In general it is the file owner who creates and maintains the files which carry his title and the protection system is based on this.

A file owner can extend some or all of his privileges to one or more nominated *part owners*. When making such a nomination the file owner gives the title of the part owner together with a status which specifies the privileges which can be enjoyed. These privileges range from permission to read from a restricted set of files to the right to create new files on behalf of the owner.

Certain privileges can be acquired by anyone who can quote an alphanumeric key specified by the file owner. A person who is able to do this is known as a *key holder.*

A prospective file user may therefore be acting simultaneously in up to four different capacities, namely: Owner, Part Owner, Key Holder and General User. In each capacity his privilege with respect to one file may be different and if he requires access to a file then it will be granted if in any of his capacities the required activity is permitted. It is the *file status* that defines the four permitted ranges of activity.

File status

Four letters, each chosen from the following set, define the status of a file, and each corresponds to one capacity in which the user might act.

- N Access not permitted.
- L Loading permitted for execution only.
- R Reading permitted for any purpose including loading.
- C Status change permitted in addition to reading.
- D Explicit deletion permitted in addition to reading and status change.
- U Updating (changing the contents) is permitted together with explicit or implicit deletion. (The latter arises if a new file is created with the same title.)
- F All activities (including change of status) are permitted.

The author of a file will usually specify the status when a file is created, although a default setting is available and file status can be changed at a later date.

File class

In addition to status the author also specifies the *class* of file. This item exists to allow the system to choose the most suitable space allocation strategy for the file. The four principal classifications are:

- A Archive file—The file will normally be held on magnetic tape.
- W Working file—The file will be held on disc where possible and, for security, a copy will be held on tape.
- T Temporary file—Disc space will be used but no security precautions will be taken.
- S System file—The file will be given preferential treatment, but will otherwise be handled as a class W file.

File deletion

We define file life as that span of time during which the file is known to the system. It is not in any way related to the existence of any one copy of a file held on one particular medium. File life is terminated in response to an explicit command, although file deletion can also take place when a new file is created

with the same title as an existing file. Both activities are subject to the status checks described above.

The performance of a filing system depends upon the volume of data held within its control, particularly where more than one storage medium is involved. Protracted file life and the undue proliferation of files must therefore be viewed with some concern. To restrict either of these without adversely affecting the value of the filing system requires a sympathetic analysis of the file owner's requirements. The user must be encouraged to make this analysis himself and a management procedure is required to ensure that the appropriate standards are being maintained. Arbitrary controls applied by the system are likely to result in abuse or undue frustration. Compromise is clearly necessary.

A system which allows the user to make post-dated requests for file deletion and status change was rejected on the grounds that, at Cambridge, the validity of a file depends more on circumstance than on time and therefore would not significantly assist the user to make a realistic evaluation of his files. In contrast, the chosen policy is to make file deletion as simple as possible by providing a variety of aids to file directory maintenance. These are included as special facilities in the command system.

Commands

The commands which are handled directly by the filing system are augmented by other macro-commands which provide more specialized facilities. The basic set is the minimum complete set necessary to provide the facilities outlined above. OPEN, CLOSE, DELETE and CHANGE STATUS provide essential file handling facilities. SET KEY and UNLOCK service the key holder system and CREATE PART OWNER and REMOVE PART OWNER service the part-ownership mechanism.

The basic set is extended so that many of the administrative features (including dump and re-load) can be carried out by conventional object programs that communicate with the central filing facility. In particular there are system calls that operate on the individual attributes of a file held by the system. Each attribute type has an associated system defined status against which all requests are checked.

File storage on magnetic tape

Magnetic tape is used for two distinct reasons; to augment the limited supply of disc space and to hold redundant copies of files so that the chances of accidental loss by system failure are reduced. Operationally the two are linked. Data are copied to magnetic tape at intervals of about 20 minutes and thereafter are protected against loss by system failure. The existence of a copy on tape leaves the system free to recover disc space when necessary simply by erasing the disc copy of a selected file. However, since the latter action reduces the level of security, magnetic tapes are duplicated.

A dump program copies to magnetic tape only those files for which there is no up-to-date dumped version. Class T (Temporary) files are never dumped. At weekly intervals the files for one owner are collected together on an archive tape. Several file owners will usually share one tape for this purpose.

If a file is deleted or updated the old version is automatically removed from the directory of files. However, since three generations of archive are kept the careless file owner has up to two weeks in which to recover an erroneously deleted file.

The file directory contains a record of the files that have been stored on magnetic tape and this is sufficient to allow automatic file recovery. In addition, it is planned to make the archive tapes available on demand so that a file owner may, for the duration of one job, have access to all his files even though the total volume may exceed the available disc space.

File reloading is batched and a file containing a queue of requests for file reloading is processed at suitable intervals. The user can file such a request and when doing so he can ask for a message to be transmitted to him upon completion. For off-line work the execution of a job may be dependent upon the availability of a file and if not available a request for reloading is added to the queue.

Disc space allocation

Each file owner is allocated a specific amount of disc space which he may not exceed. The allocation is split into space for ordinary files and space for temporary files. If a filing attempt for an ordinary file fails through lack of space then the file will be reclassified as temporary and a second attempt will be made.

The total space allocated exceeds that available and thus it is possible for space to be exhausted even though no user is in error. In this case all the files for one user are off-loaded. The user affected in this way is chosen in strict rotation from all those not active at the time. When a file is off-loaded the disc copy is erased as soon as it is known that a copy exists on magnetic tape. Temporary files are lost by the off-loading process.

System files and other files which are in general use are classified as S and are never off-loaded. Each user is given a (usually zero) space allocation for his

class S files; all users are treated equally.

Protection against system failure

File dumping takes place every 20 minutes and has already been described. Each dump tape contains sufficient information to allow the file directory to be reconstructed. The restart process recovers files from dump tapes and archives ad-lib until recovery is complete. The first files to be recovered are those owned by the system which are held on the appropriate archive; thereafter recovery proceeds simultaneously with general use of the filing system.

Many system failures do not result in loss of information on disc and a recovery procedure is available that retrieves as many files as possible on the assumption that the file was not over-written arbitrarily. To make this possible the methods used to update the disc control map and the file directory must failsafe and this it is hoped has been achieved by carefully sequencing the operations which take place during the file creation and deletion. In a system in which overlapping is extensively employed, care is required to ensure that the official recognition of a change in state is withheld until that state has been established.

CONCLUSION

A number of implementation points are worth noting because of the extent to which coding and maintenance are thereby simplified. A privileged but otherwise conventional user program handles file access and the various necessary interlocks. The file processing and disc control are again separate but part of the supervisor. All other facilities (including dumping and file recovery) are carried out by conventional user programs. One file format is used for all dump and archive tapes and all file directories are held as files in their own right. The interlock requirements are thus minimized and by holding a detatched map the timing considerations which are essential to a successful recovery system are simplified.

ACKNOWLEDGMENTS

The work described in this paper is part of the program of research and development being carried ou at the University Mathematical Laboratory, Cambridge under the direction of Professor M. V. Wilkes F.R.S. The early stages of the basic supervisor work were carried out in collaboration with International Computers and Tabulators Ltd., subsequently the project has been supported by the Science Research Council.

REFERENCES

1 T KILBURN R B PAYNE D J HOWARTH
The Atlas Supervisor
Proc E J C C n 279 1961

2 A G FRASER J D SMART
The COMPL language and operating system
Computer Journal vol 9 p 144

3 R C DALEY P G NEUMANN
A general-purpose file system for secondary storage
Proc F J C C p 213 1965

GIM-1, a generalized information management language and computer system

by DONALD B. NELSON, RICHARD A. PICK AND KENTON B. ANDREWS

TRW Systems
Redondo Beach, California

The current demand for more comprehensive information systems had created an increasing need for generalized information management. Generalized information management denotes a complex of interrelated information capabilities encompassing both human functions and the operation of automatic devices which supplement and extend human capabilities. The methodology and techniques for such generalized information control have been defined and are exemplified in GIM-1, a generalized information management language and computer system now being implemented at TRW Systems. With the use of GIM-1, the problems inherent in defining more comprehensive information systems can be completely and economically resolved.

The definition of a system for such information management requires a communication network of many remote stations and a central complex of one or more computers. It also must accommodate natural language queries and multiple technical vocabularies. Additionally, such a system must provide automatic correlation of information both within and between data files, complete as well as selectively limited information security, and automatic controls for data reliability, data conversions and the synthesis of data as a function of other data. All of these requirements for comprehensive information transfer can be satisfied by the use of GIM-1.

GIM-1 permits natural language access to data list information stored in a random bulk storage device which also is available to other computer programs, and the remote user may select any available equipment in the entire network for an information output. Although GIM-1 is general to information management needs, its design accommodates the particular information requirements of any one application area with special purpose efficiency in computer operating times. GIM-1 is essentially machine independent, and the calculated addressing scheme permits retrieval of any required data from random storage in a single access.

The GIM-1 language is limited natural English, and the formats and rules for remote inputs are both simple and very general. The language processors, together with the use of dictionaries, permit inputs to be stated directly in the technical terminology natural to each application area, and also provides the user with plain language outputs. The GIM-1 language uses the lineal format natural to prose text, accepts any number of variable length words, and permits a limited freedom of word order. Minimal vocabulary prohibitions to ensure uniqueness apply only to the data list identifiers and are guaranteed by automatic language audits, but no vocabulary prohibitions are imposed on the information values. Additionally, the GIM-1 language permits the definition of any number of identifer synonyms, and provides a large selection of input conditionals which may be used to limit outputs to only relevant information.

In GIM-1, the "master" dictionary contains the data list identifier nouns, and the "user" dictionary associated with each data list contains the attribute identifier nouns. The master dictionary also contains the process identifier verbs and the language connectives; and, in each of these dictionaries, any number of synonyms may be defined for each identifier. The connective words include all conjunctions, prepositions, articles and special symbols specified for possible use in language inputs. The definition

of these connectives, then, includes the designation of relational operators and other conditional words for limiting information searches to only relevant data. Therefore, the specification of connectives will dictate operational procedures not only for the language preprocessor but also for the automatic correlation of data and the numerous data processors within GIM-1. Additionally, a limited natural syntax has been defined for GIM-1, since any completely natural syntax would require a probabilistic processor contradictory both to practical efficiency and to the absolute identifications required for updating data lists with complete accuracy.

The GIM-1 data base consists of many different data lists, and all GIM-1 information is stored in data lists. Each data list is identified by a DATA LIST I. D. and consists of many different items of information; and each item consists of an ITEM I. D. followed or not by relevant information values. All information values, then, are stored in items within data lists, and each VALUE is identified by one of the many attributes defined for each data list. Each attribute is identified by an ATTRIBUTE I. D. and any number of attributes can be defined for the identification of all possible data list values. However, since items contain only relevant values or none at all, the usual document has values for only a few of the many attributes defined for a data list.

The standard format for organizing GIM-1 information, then, has a column heading for each of the following four information elements:

Information element	Mnemonic Code
DATA LIST I.D.	D
ITEM I.D.	I
ATTRIBUTE I.D.	A
VALUE	V

An example of information organized in accordance with such a format is illustrated in Figure 1.

Since a data list may have virtually unlimited attributes which may be used or not in any given item, a user is neither restricted to a small set of fixed identifiers nor required to assign a value to each identifier for every item. For example, assume all references for transporation systems are itemized by library code, with each reference containing such standard information as TITLE, AUTHOR, ABSTRACT, etc. In addition to these standard attributes, however, much greater indexing depth can be achieved by also defining many special attributes such as GROUND EFFECT MACHINES, HELICOPTERS, MONORAILS, TOTAL COST, and PRINCIPAL CITY. These extra attributes, then, define special information which may or may not be relevant to any one transportation system reference. For instance, the majority of references may include absolutely no information for these extra attributes and, therefore, no extra values would be stored. However, for any item which may contain such special information, these extra attributes would enable an indexer to describe the reference item in much greater depth. Provided such attributes were defined, then, a GIM-1 user might initiate the following input:

> LIST THE TITLE AUTHOR AND ABSTRACT OF EVERY TRANSPORTATION SYSTEM REFERENCE WITH THE PRINCIPAL CITY "LOS ANGELES" AND THE GROUND EFFECT MACHINES "AVC-1" OR "HOVERCRAFT"

Furthermore, the organization of GIM-1 information permits the remote user to add new attributes for a data list whenever he may wish, and without any effect on the existing data list information. For instance, an indexer might encounter a transportation system reference which included a description of the net change in smog levels during trial tests of some proposed system. Assuming all of the existing attributes to be unsuitable for a description of this information, the indexer might wish to add the new attribute HYDROCARBONS. This new attribute then would be available for possible use in any old or new transportation system reference.

The four elements within the standard format for GIM-1 information can be considered both as an attribute value DATUM having three IDENTIFIERS (Figure 2), and as a set of two dyads, with each dyad having one DATUM and one IDENTIFIER (Figure 3).

Consideration of the standard format as a sequence of two dyads of information is a concept of particular importance. Together with the automatic correlation of GIM-1 data, both within and between data lists, this concept permits the definition of extensive and very complex information formats. The construction of an extended data format by a unidirectional chaining of dyad units is illustrated in Figure 4.

As an elementary example of data format extension, assume the unidirectional construction in Figure 5 as defining both the retrieval and update-add correlation between the attribute AUTHOR in the ANEW DOCUMENT data list (defined in Figure 1) and another data list identified by TECH/AUTHOR.

In Figure 6, therefore, any value for AUTHOR automatically will be also an item I. D. for TECH/

DATA LIST I. D.	ITEM I. D.	ATTRIBUTE I. D.	ATTRIBUTE VALUES
ANEW DOCUMENT	NADC-AX-6123	TITLE	MOD 3 SIMULATION DESIGN
		AUTHOR	JOHN DOE
			WALTER SMITH
		SOURCE	DDC 457 321
		DESCRIPTORS	P-23 SIMULATION
			ADX AIRCRAFT SIMULATION
		DATE	12 NOV 65
		ABSTRACT	DESIGN FOR SIMULATION OF JULIE AND JEZEBEL EXERCISES WITH P-23 AIRCRAFT
	NEL-251367	TITLE	S012 REPORT
		SOURCE	NEL
		REMARKS	ORDERED ON MAY 7, 1966
	NOL-25682	TITLE	BALLISTIC PERFORMANCE
		AUTHOR	PETER BROWN
		SOURCE	DDC 123 896
		DESCRIPTORS	X503 TRAJECTORY
			MISS DISTANCE

Figure 1 – Example of GIM-1 data organization

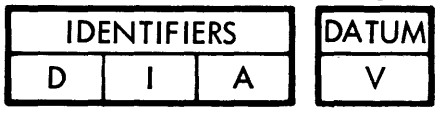


Figure 2 – Standard data format A

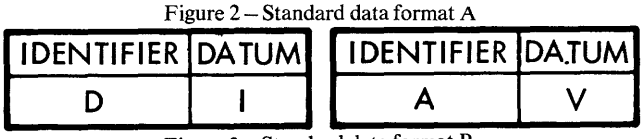


Figure 3 – Standard data format B

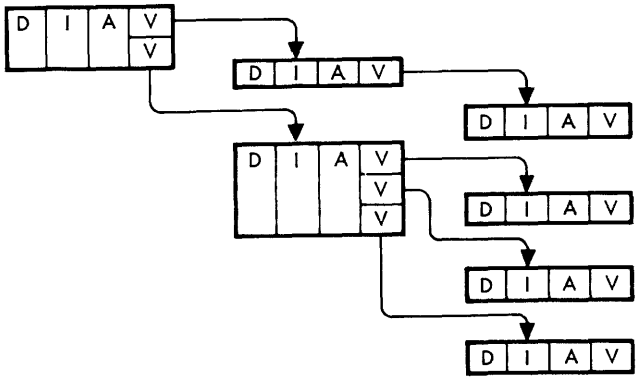


Figure 4 – Extended data format

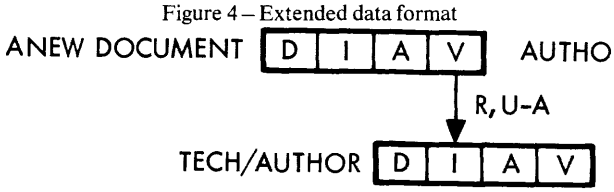


Figure 5 – Example of extended data format

AUTHOR, although each item I.D. for TECH/AUTHOR is not necessarily also a value for AUTHOR automatically also will include, as associated data, all the information stored under the relevant item I. D. for TECH/AUTHOR. However, any retrieval of an item I. D. for TECH/AUTHOR will not include any associated data.

The elementary correlations in Figures 4, 5, and 6 are constructed by unidirectional chaining of the values of an attribute in one DIAV unit with the item I. D. in another DIAV unit. However, many other types of automatic correlation also are available to the GIM-1 user. For instance, "Christmas Tree" interrelationships between different items within the same data list require bidirectional chaining of the values of an attribute with the item I. D. in a single DIAV unit. Additionally, some commonly required correlations are considerably more complex. For instance many interrelationships require correlations between the values of two attributes in different data lists, together with automatic item identifications. This type of correlation is illustrated in Figure 7 as an elementary construction with only two unidirectional correlatives and with only one correlative defined for the values of an attribute.

In Figure 7 then, each update of a value in D_1 A_3 requires the automatic update of an item I.D. in D_2. Similarly, each update of a value in D_1A_1 requires the automatic update of a value in D_2A_4. However, the automatic update of D_2A_4 further requires the automatic identification of each item I.D. in D_2 which also is defined as a value in D_1 A_3. The automatic creation of such "bridge" values

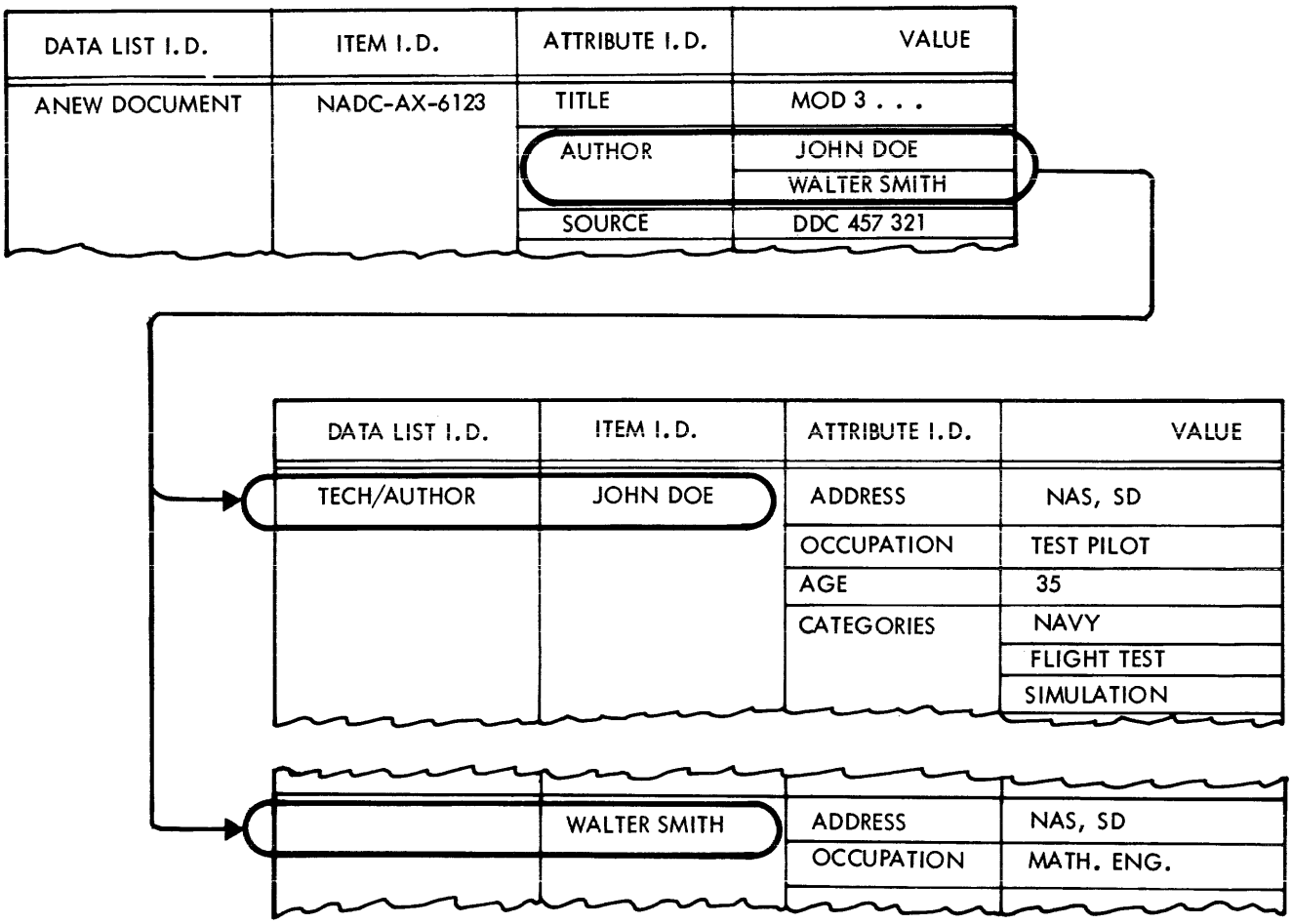


Figure 6 – Example of extended data format

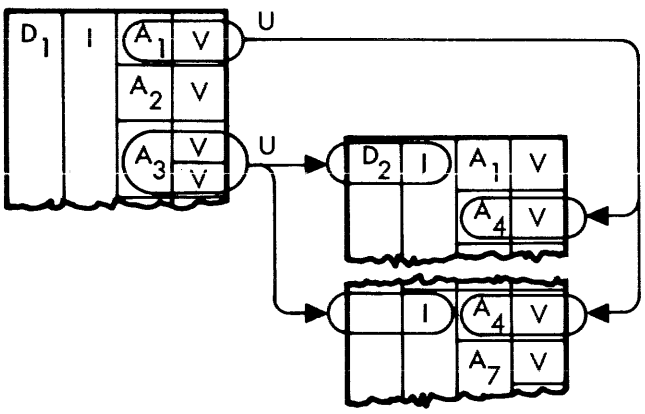


Figure 7 – Example of data list correlation

between data lists, then, is required for any correlation between the values of two attributes in different data lists. Although these few examples of automatic correlation are elementary, they are sufficient to indicate the flexibility provided the GIM-1 user in defining extensions of the standard format for organizing information.

GIM-1, then, permits the user to specify very complex data list interrelationships for automatic correlation. Only a simple dictionary code enforces any defined correlation for all subsequent inputs, and GIM-1 processes all interrelated information with total relevance and with complete accuracy. The several correlations available for specification by the GIM-1 user include:

- The automatic retrieval of associated information stored in other data lists.
- The automatic updating of correlated information stored in other data lists.
- The automatic updating of cross-indexed data lists.
- The automatic calculation of data values as a function of other stored values.
- The automatic coupling of unit values in associated sequences of multiple values.
- The automatic creation of "bridge" values for correlation between data lists and items.
- The automatic updating of all relevant information in "Christmas tree" data lists.
- Full "Christmas tree" searches for retrieval and counting requests.
- The automatic retrieval and updating of non-redundant information stored only in another data list.

Among these several optional capabilities defined for user convenience, the automatic updating of information in correlated items and data lists is of particular importance.

For example, a user might require optional access to information by classifications unique to his own technical area, while the technical library might wish to file new information only by DDC index number. These apparently different user needs are not contradictory, however, since both requirements can be accommodated by the automatic updating of correlated information. Assuming the information for each DDC index number is defined to include such attribute identifiers as TITLE, AUTHOR, ABSTRACT, SOURCE, TECHNICAL AREA and DESCRIPTORS, and also assuming the relevant correlation codes are stored in the dictionaries, the GIM-1 system would update not only the DDC data list stated in the technical library input, but also automatically would update any number of correlated data lists and items defined by the dictionary codes and the values stored under the encoded identifiers. This complex capability, therefore, eliminates not only the need for complex intervention by the user but also the probability of human error, and ensures the complete accuracy and relevance of interrelated information.

GIM-1 further accommodates the remote user by providing complete as well as selectively limited information security, automatic value conversions and complex format audits of input values. Additionally, the executive control system provides priority processing and automatic record-keeping of all transactions. The standard data processing capabilities of GIM-1 include the retrieval of stored information; the counting of information values; the deletion, addition or changing of stored information; the remote initiation of new dictionaries and data lists and a special report processor for generating special output formats.

As exemplified by GIM-1, then, generalized information management is based on a new and interdisciplinary approach to the total problem of information, and is a comprehensive complex of interrelated capabilities. As indicated by its name, however, GIM-1 is only an initial competence with many extensions of present capabilities, as well as new capabilities, planned for the future. At present, for example, the automatic updating of information between data lists provides a selective dissemination of information, but only within the GIM-1 data bank. However, the rapidly growing need for selective dissemination requires an external distribution; and, because of the construction of functional group interest profiles, the impetus for the flow of information rests with the system rather than with the user. Therefore, in a future version of GIM-1, the present selective dissemination of information will be extended to include automatic decision-information outputs to all users. Similarly, the increasing demand for international information transfer requires a system which accommodates multiple languages, as well as multiple technical vocabularies, with direct access to mutually shared data banks without language translation. However, GIM-1 already accommodates multiple technical vocabularies with direct access to mutually shared data lists without vocabulary translation. Additionally, GIM-1 is both machine independent and language dependent. Therefore, in a future version of GIM-1, the present natural English language will be supplemented by Russian, French and German.

This description of GIM-1 has been brief, and the summary of its capabilities gross. As an example of generalized information management, however, it has been sufficient to indicate the methodology and some of the techniques for resolving the problems inherent in designing more comprehensive information systems. Also, it has indicated the need for integrating the definitions of company information standards and management responsibilities with the specification of a communication network and the software control system. And hopefully, this brief description of GIM-1 has been sufficient to verify generalized information management as a new and powerful tool for supplementing and extending present capabilities in information systems.

Inter-program communications, program string structures and buffer files

by E. MORENOFF and J. B. McLEAN
Rome Air Development Center
Griffiss Air Force Base, New York

INTRODUCTION

Receptiveness to change is one of the most important attributes of a programming system. There are obvious advantages associated with being responsive to changes in a system's environment without being required to essentially redesign the system. These environmental changes may take the form of variations in the functions to be performed by the system or in the applications to which the system is put. They may be reflected in modifications of the content or structure of the data which the system is to process. They may even appear as changes in the number, type or function of the equipments comprising the computer within which the system is operated.

The introduction of program modularity is one of the most common approaches to designing programming systems capable of evolutionary growth and modernization. Traditionally, modularity is associated with the division of a program system into its major components, which in turn are divided into their subcomponents, and so on. The degree to which this can be actually reduced to practice, however, is a function of the facilities available for combining the program modules into larger programs. Factors which must be considered are the extent to which inter-program dependencies exist, the nature of the mechanism available for resolving such dependencies, and the time at which they are resolved.

If the program modules could be treated as completely separate entities, wholly independent of one another, they could be used as program building blocks to be arbitrarily combined to form larger programs. This generalization leads to the notion of a pool in which all program building blocks are maintained, available for use in larger programs to perform user-oriented functions. The application of Program String Structure techniques[1] allows such a generalization to be made, and hence allows for the realization of the notion of a program building block pool.

The most essential feature of the Program String Structure concept is the introduction of Buffer Files through which all inter-program communications are achieved. Although the primary motivation for the development of this Buffer File mode of operation was to provide a program linkage mechanism which could be used as part of a Program String Structure, several very important benefits can be realized by its application outside the Program String Structure environment. It is the intent of this paper to examine these benefits in terms of the unique characteristics of the Buffer Files which make their realization possible.

Program string structures

The Program String Structure is based on the fundamental principle that a set of programs can be used collectively without actually being combined or integrated. This collective use of programs is realized by providing a mechanism for moving the data from one program in the set to another. Each program in the set is afforded, in turn, an opportunity to operate on the data in some prescribed manner. The mechanism is sufficiently general to allow for the collective use of totally independent programs.

A program in the context of the Program String Structure, is a physically identifiable unit through which streams of data are passed. The program accepts one or more input data streams, operates on them in accordance with the logic implicit in the program, and then produces one or more output data streams. All programs are maintained in a program pool. Figure 1 illustrates the logical representation of a program.

The primary characteristic of the Program String Structure is the introduction of a Buffer File in the path of each stream of data flowing from a output of one program to an input of another program. A program is designed so that its input data streams are taken from Buffer Files and its output data streams go to Buffer Files. The program need not be concerned with where the data found in its input Buffer File comes from (or how or when it got there), nor where

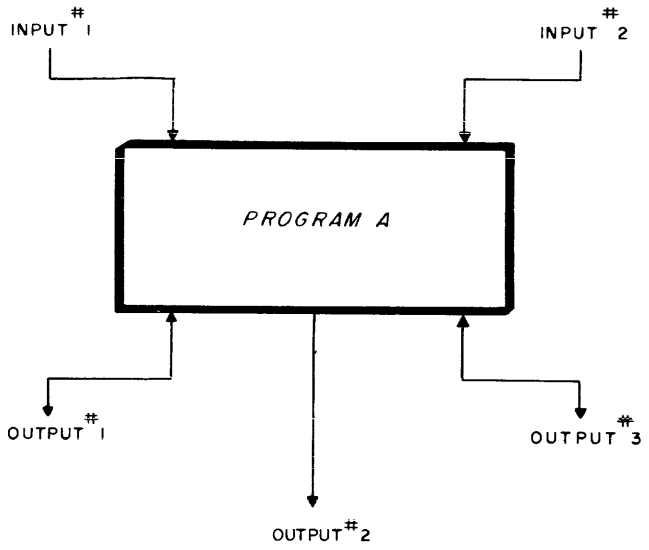


Figure 1 – Logical representation of a program in a program string structure

the data it places in its output Buffer File goes to (or when it will get there or what will be done with it). The operation of a program is completely asynchronous with respect to any other program with which it may be collectively used. The synchronization of the program with concurrently running programs and the coordination of access to common data between itself and concurrently running programs, are automatically performed by the Buffer File linkage mechanism. The Buffer Files thus completely isolate a program from the programs with which it is linked. Figure 2 illustrates the logical representation of the use of Buffer Files as a program linkage mechanism.

The use of Buffer files is a necessary but not sufficient condition for allowing sets of totally independent programs to be used collectively. Each program assumes a format and structure for all data on which it operates which is inherently embedded in the logic and code of the program. Hence, a program expects the data at each of its inputs to be of a specific format and structure. The program, in turn, generates data at each of its outputs in a specific format and structure. Consequently, the Program String Structure must provide a means for insuring that that data reaches a program's input Buffer File in a form required by the program.

To insure data is properly formatted and structured when it reaches a program's input Buffer File, the format and structure specifications of the output and input data streams which are to be linked are automatically compared and then adjusted for reasonable format discrepancies. This implies that the descriptions which define the format and structure of each of the output and input data streams of a program must be explicitly stated and available for use by that portion of the Program String Structure which compares and adjusts data formats. Reasonable adjustments can be accomplished by the automatic insertion of a generalized reformatting program in the data stream. The inputs to the generalized reformatting program are the input and output data stream specifications, and the data in the streams to be reformatted. The output of the generalized reformatting program is a stream of data in the desired format. The generalized reformatting program is a type of report generator program. Since the outputs and inputs of the programs being linked are isolated from each other by Buffer Files, the introduction of the reformatting program in no way influences the execution of either of these programs. Figure 3 illustrates the logical representation of the use of the reformatting program. The case is the same as shown in Figure 2, except that it is assumed here that the format specification of the third output of Program A is not compatible with the first input of Program C and must be adjusted by the use of the Generalized Reformatting Program.

Another characteristic of the Program String Structure is the introduction of a variable time frame over which program inputs are actually bound to the data on which the program will operate. At the time a program is designed, the programmer assigns symbolic names by which each input and output is internally referred to by the program. At this time, he also specifies the formats and structures of the data which are required by the program at these inputs and outputs. At linkage definition time, the user specifies which programs are to be used collectively, and which outputs are to be linked to which inputs. At this time, the user may also bind some of the free-standing inputs to data on which the linked programs will operate. A free-standing input is one which is not connected to any other program except the one for which it is an input. The user may elect to defer the binding of some of the free-standing inputs to data until the time the linked programs are to be executed. If such is the case, these unbound free-standing inputs may be specified as part of the user's request to have the set of linked programs executed. It is possible, however, that not all the free-standing inputs have been bound to data at the time the execution of the set of linked programs has been initiated. If, as a result of the logic of a program and the nature of the data being processed, no data reference is made to an unbound free-standing input, then the fact that the free-standing input has not been bound does not affect the processing of the set of programs. If, however, a point is reached in one of the programs where a request is made for data from a

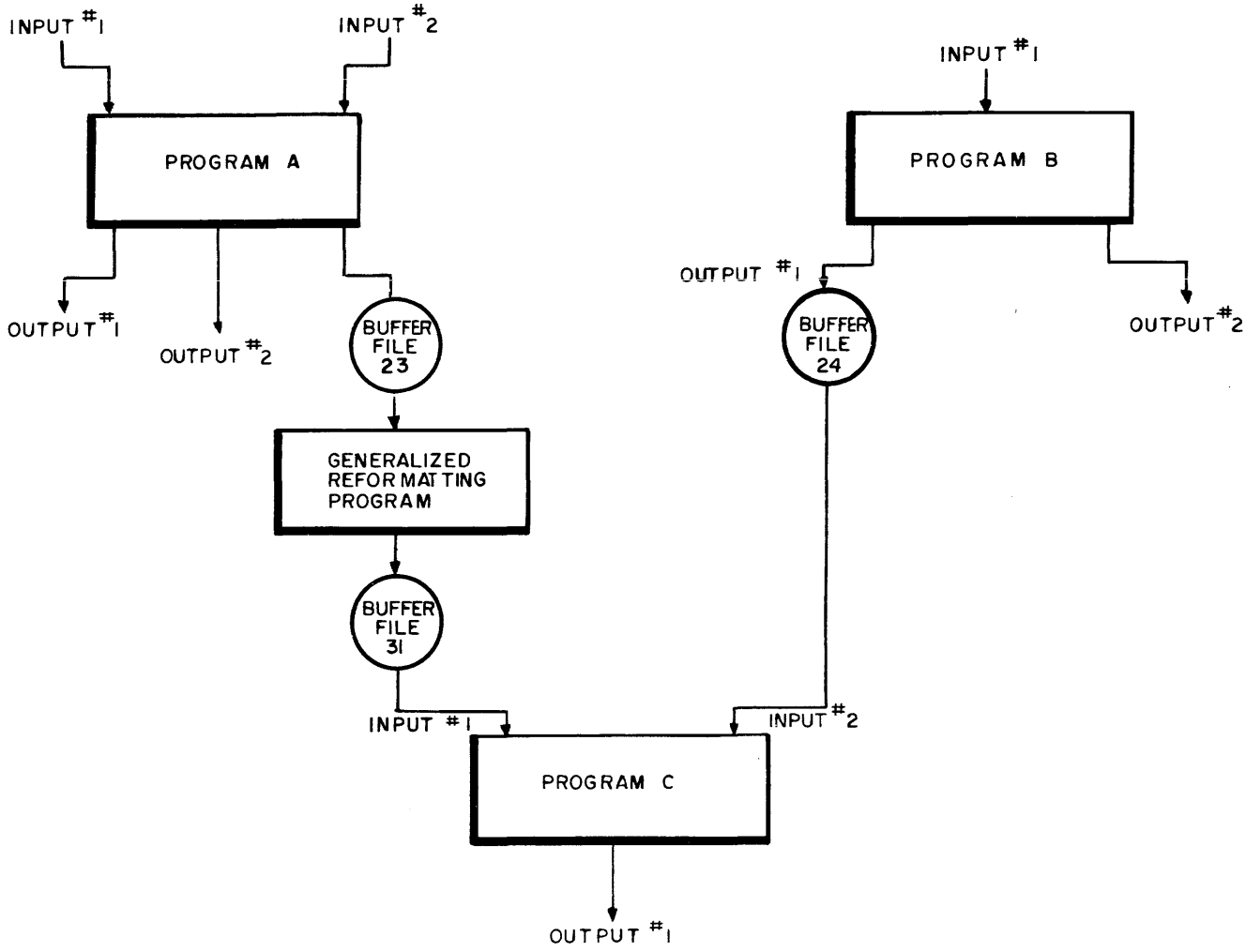


Figure 2—Logical representation of programs linked via buffer files

previously unbound free-standing input, then the linkage mechanism causes the execution of that program to be delayed until the monitor system can request the user to supply the missing binding information, and such information is provided by the user.

Finally, programs may be recursively defined in a Program String Structure. This is possible to the extent that a program representing a set of linked programs can itself be treated as a program building block and maintained in the program pool to be used collectively with other programs. Figure 4 is an illustration of the recursive definition of a program in which Programs R, S and T are linked to yield Program A, initially shown in Figure 1.

The primary result of the use of Program String Structure techniques is the elimination of the need for pre-planning on the part of the programmer during the program design and implementation phase. The programmer may be totally indifferent to the characteristics of other programs with which the program he is preparing may operate; the variations in structure and source of data on which his program may be called on to process; and any timing or coordination considerations with respect to other programs which may be concurrently executed. The programmer is free to design a particular program to optimize that program's performance solely with respect to the function being implemented.

The elimination of the need for pre-planning means that a set of programs can be designed, coded, debugged and modified by different programmers at different points in time and still be capable of being arbitrarily linked. This is the essential characteristic of a true building block approach to the design of large programming systems.

The construction of large programs by linking together as many already existent programs as possible (and creating new programs only when necessary) materially shortens the overall program preparation

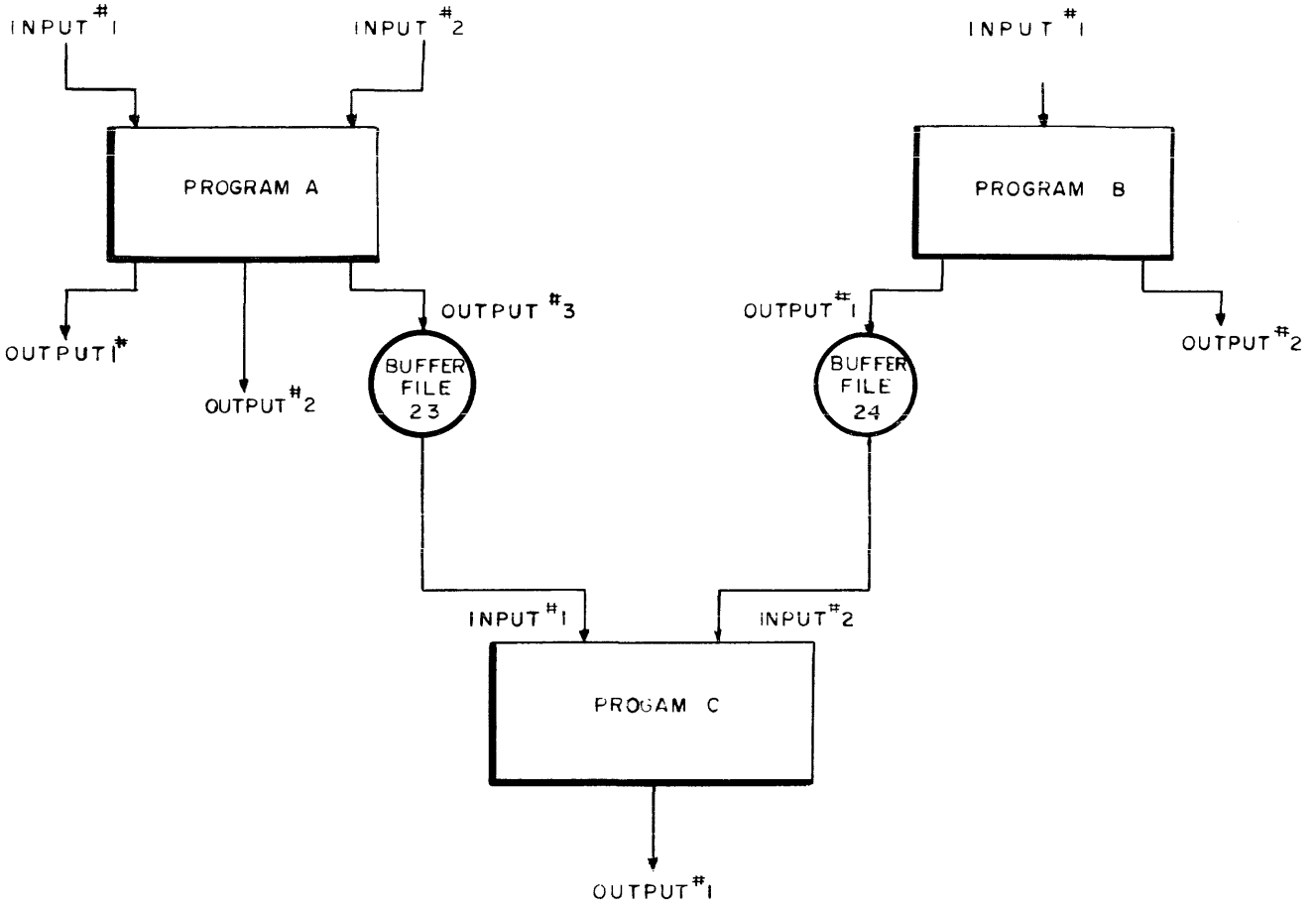


Figure 3 – Logical representation of automatic reformatting capability

time at the expense of an increased overhead associated effecting the linkage of the smaller programs into larger ones. In environments in which many of the programs are executed on essentially a "one-shot" basis, such a trade-off seems reasonable. Should any of these programs eventually be used on a production basis, however, thus tending to increase the significance of the overhead, the structures of these programs can be modified by removing the inter-program linkages. Changing the overall program from a set of linked programs to a single program eliminates the overhead previously associated with the program. The program restructuring, however, can be done at a convenient time with respect to such considerations as the availability of programmers, existing work loads and priorities. In the meantime, the desired function can be performed as a collection of smaller programs.

It is important to note that although the use of Program String Structures eliminates the need for preplanning, the programmer retains the ability to introduce pre-planning in the design of a set of programs to any degree he wishes. Thus, in the design of a set of programs which will be frequently used, the programmer may elect to carefully consider questions of inter-program communications in order to improve the performance of the set of programs. The programs making up the set, however, may still be placed in the program pool and collectively used with other programs to perform other functions as well.

Buffer files – principles of operation

A conceptual representation of a Buffer File is shown in Figure 5. The Buffer File shown is employed in the linkage of the third output of Program A to the first input of Program C, as illustrated in Figure 2. The data in the Buffer File is organized in fixed length units called blocks. There are M computer words per block and N blocks per Buffer File.

Only a unidirectional flow of data through a Buffer File is permitted. As shown, Program C may only read data from the Buffer File and Program A may only write data to the Buffer File. Once the data has been transferred from Program A to the Buffer File, that data can no longer be accessed by Program A. Similarly, once Program C has read data from the Buffer File, that data is erased from the Buffer File.

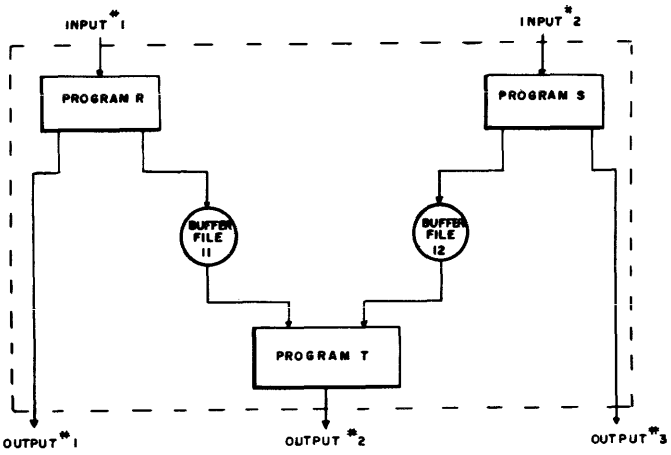


Figure 4 – Logical representation of a set of programs linked to form a new program

In order that programs using a Buffer File be independent of both parameters M and N, the Buffer File closes around itself, forming a ring structure. Addition is performed on a modulo N basis, that is (N-1) plus one equal zero. Two markers are associated with each Buffer File, one indicating the name of the block to which data is to be written and the other from where the data is to be read. In Figure 5, the Write Buffer Marker is advanced each time a block of data is placed in the Buffer File by the third output of Program A, and the Read Buffer Marker is advanced by one each time a block of data is read from the Buffer File by the first input of Program C. If the Read Buffer Marker and the Write Buffer Marker should coincide and the two markers have recycled the same number of times, no further data is available in the Buffer File for Program C to read. Hence, any read operation attempted by Program C under this condition must be inhibited. Similarly, if the Read Buffer Marker and the Write Buffer Marker should coincide and the Write Buffer Marker has recycled one more time than the Read Buffer Marker, no further space is available for Program A to write in the Buffer File. Hence, any write operation attempted by Program A under this condition must be inhibited. Finally, since only a unidirectional flow of data is permitted through the Buffer File, any attempt at any time by Program A to read data from the Buffer File or Program C to write data to the Buffer File must be inhibited. As soon as the presence of any of the inhibit conditions is detected by the linkage mechanism, control is immediately transferred to the computer monitor system, thus inhibiting the execution of the read or write operation.

During the design and implementation phase of a program, the programmer symbolically defines the input and output Buffer Files to be used by the program. It is not, however, until that point in the execution of a program is reached where a request for the creation of a Buffer File that the Buffer File is generated. In this manner, although a programmer may specify multiple outputs in this program to cover mutually exclusive conditions. Buffer Files are created only for those outputs which the program actually selects on the basis of the data being processed.

The actual linkage between an output of one program and an input of a second program is effected through the use of a Buffer File Control Word (BFCW) generated by the linkage mechanism. One BFCW is maintained by the linkage mechanism for each Buffer File established. The BFCW contains such information as the positions of the Buffer File's Read and Write Buffer Markers, the Buffer File's size, location and a series of indicators reflecting its condition and status.

Either of the two programs to be linked via a particular Buffer File may be the first to reach that point in its execution where the request is made for the creation of the Buffer File. In response to this request, the linkage mechanism generates a BFCW and establishes a correspondence between the BFCW and the symbolic name by which the Buffer File is referred to in the program. When the other program to be linked makes a request for a Buffer File, using a symbolic name equated to the symbolic Buffer File reference in the first program at linkage definition time, the linkage mechanism establishes a correspondence between that symbolic name and the same BFCW. The method by which the correspondence between the BFCW and the symbolic Buffer File references in the programs to be linked is instrumented, is a function of the particular environment in which the linkage mechanism is implemented.

The following types of commands are typical of those which could be used in a programming language to control the Buffer File mode of operation:

OPEN BUFFER FILE	* ALPHA, M, N
WRITE BUFFER FILE	* ALPHA
READ BUFFER FILE	* ALPHA
COMPLETE BUFFER FILE	* ALPHA

The OPEN command is used by a program to request the linkage mechanisms for a Buffer File. In-

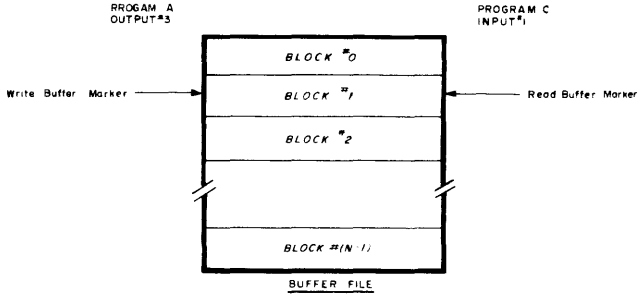


Figure 5 – Conceptual representation of buffer file operation

clusion of M and N enables the programmer to specify preferred block and file sizes, respectively. The actual transfer of data to or from a Buffer File is requested by use of the WRITE and READ commands respectively. The COMPLETE command is used to make the linkage mechanism aware of the end of the requirement for a Buffer File.

Although the flow of data through a Buffer File is unidirectional, bidirectional communication between two programs can be effected by the establishment of separate Buffer Files whose direction of data flow are in opposite directions. Figure 6 illustrates the logical representation of the use of Buffer Files for bidirectional inter-program communications.

Figure 6 also illustrates another important advantage derived from the use of the Buffer File linkage mechansim. In the event that Program Module Y does not yield satisfactory results (accuracy, speed or core space), it may be replaced by a different program module Y′, simply by defining a linkage between W and Y′, and requesting the execution of the new set of linked programs. No re-assembling is necessary to accommodate this change, which can be effected in an "on-line" mode of computer operation.

Implications of the use of buffer files

The characteristics exhibited by the Buffer Files are such that their application as separate entities apart from Program String Structures, for which they were initially conceived, has considerable merit.

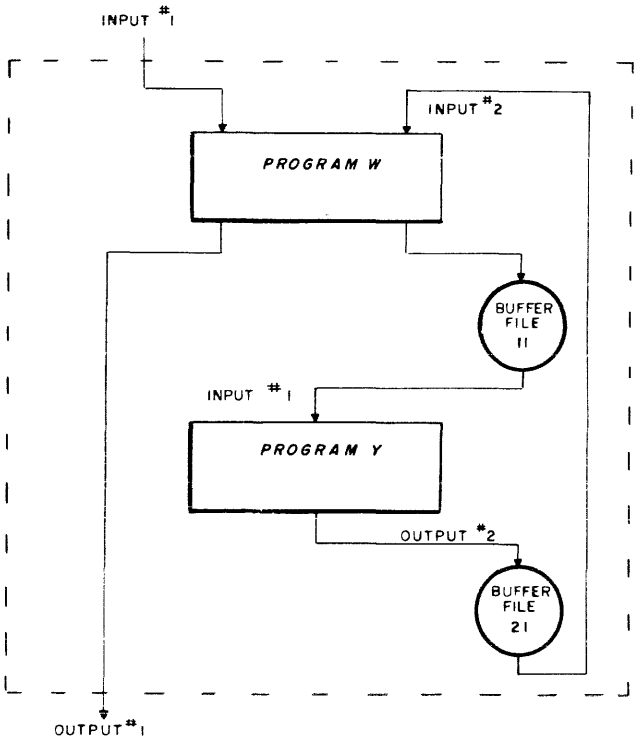


Figure 6 – Bidirectional inter-program communications

Consider first, the ability of a Buffer File to synchronize the execution of concurrently running linked programs. Program synchronization is effected by regulating the streaming of data through a Buffer File from one program to another. A program writing data to a Buffer File cannot proceed beyond the point in its execution when it is necessary to place data into the Buffer File and there is no room for additional data in the Buffer File. Similarly, a program reading data from the Buffer File cannot proceed beyond the point in its operation when it requires data from the Buffer File and there is no additional data in that Buffer File. The temporary delay of a program, either waiting for additional data in its input Buffer File or for additional space in its output Buffer File, is effected automatically by the linkage mechanism which transfers control to the monitor system each time the program attempts to do a read or write operation to its Buffer Files under such conditions.

With the exception of this synchronization, which takes place as the data passes through the Buffer File, the two programs can be executed completely independently of one another. The computer monitor system may arbitrarily transfer control to either of the two programs, which then resumes its execution at the point at which control is returned. Should the processing of that program still have to be delayed, any further attempt to do either of these blocked operations immediately returns control to the monitor system which may then select another arbitrary program to be operated on.

In this regard it should be noted that the monitor system can regulate the rate of execution of a program by adjusting the sizes of the program's input and output Buffer Files. Changing the size of the Buffer File moderates the flow of data through a program and hence also the rate at which the program may operate on the data.

The Buffer File mode of operation thus affords the computer monitor system a basic mechanism upon which more complex and sophisticated scheduling and resource allocation algorithms can be built. The linkage mechanism provides the means for both regulating the speed at which a program may be executed and automatically withdrawing control from a program when it is unable to proceed.

The streaming of data through a Buffer File from one program to another leads to a mode of program execution which is particularly useful in "on-line" computer systems. This mode of program execution involves the passage of partial results to a second program before the first program has completed its operation. The utility of being able to pass partial results between programs in such environ-

ments is related to the ability to make a part of the results available to the user prior to the completion of the entire program.

Buffer Files may also be used for conveying information not generally thought of as flowing between programs. They can, for example, be used to convey change of state information from one program to another. The Buffer File block size is chosen on the basis of the quantity of information to be passed between programs. For this application a pair of Buffer Files will generally be required so that bi-directional communication can be effected. When the internal state of either of two programs is altered, the nature of the change can be entered as a block of data into the output Buffer File associated with that program. This causes the Write Buffer Marker in the corresponding BFCW to be advanced by one. If it was assumed that the positions of the Write and Read Buffer Markers were initially equal, then the change in state of one of the programs causes this marker equality to be destroyed. The fact that there has been a change in state of one program can readily be sensed by the other program which then and only then is permitted to read a block of data from the Buffer File. This causes the Read Buffer Marker in the corresponding BFCW to be advanced by one. Hence, the detection of the change in state of the first program by the second program causes the marker equality to be restored.

The second program can determine the nature of the change in state of the first program by the examination of the data in the block it has read from the Buffer File. The transfer of program state information is thus effected, not by the use of conventional "common" areas shared between programs, but rather by the use of "common" areas which have been separated into two distinct areas. The first area can only be read from by the first program and written to by the second program, while the second area can only be read from by the second program and written to by the first program. This method of exchanging state information between programs not only provides a mechanism for synchronizing the execution of two otherwise asynchronously executed programs, but also eliminates the internal program housekeeping which would normally be required for coordinating the accesses and the sequences of such accesses of the programs to the program state and condition information.

The concentration of flags, indicators, counters and registers referenced by a program into what amounts to "communications" blocks improves the efficiency with which such information can be passed between programs via Buffer Files. The use of these communications blocks can also significantly improve the performance of computer systems incorporating "page-turning" capabilities. By collecting these program references, the number of pages which have to be turned into and out of core in order to execute a program can be materially reduced from the number of pages required if such references were distributed throughout the program over many different pages.

The Buffer File mode of operation provides a perfectly general mechanism for effecting inter-program communications. The mechanism can be implemented either entirely within a computer processing unit to permit the linking of programs in a multi-programming environment, or in multiple computer processing units to permit the linking of programs in a multi-computer environment. The ability to synchronize the execution of otherwise independently executed programs is at least as important in the multi-computer environment in which the programs actually are executed in parallel, as it is in the multi-programming environments in which the execution of multiple programs only "appears" to be in parallel.

In a multi-programming environment, the Buffer Files compete for core storage in the same manner as would any program or data area in the system. If the environment has a hardware "page-turning" capability, then a Buffer File is as likely to be "turned out" as are the programs for which it is a data source or sink. The availability of page-turning hardware should, however, improve the overall efficiency of the Buffer File operation by reducing the overhead associated with insuring that the Buffer Files are in core when needed. In this regard, the performance of the page-turning hardware should be enhanced by the use of fixed-length blocks and files as part of the Buffer File operation.

In a multi-computer environment, the Buffer Files may reside in some random access storage device jointly accessible by any of the processors. Since the inter-program communication is effected by the linkage mechanism, which in reality is part of the monitor system of each of the processors, the programs look and act the same as if they were all sharing a single computer processing unit. The request for Buffer File operations is translated by the monitor system to calls on the random access storage device. As far as the programs are concerned, the actual location of the Buffer Files is unimportant.

The Buffer File mode of operation is also useful in environments where special devices are integrated with the computer processors to perform certain functions more efficiently or with greater speed. The mode of linking the programs using such devices to programs using the conventional computer processor should be such that functions employing special

devices can continue to be performed in the event that the special device is inoperative. The Buffer File mode of operation is particularly well suited to satisfying this requirement because of the insulation which the Buffer Files provide to a program.

Other techniques

The development of the Program String Structure as a means for achieving inter-program communications is a logical extension of the current technology in this area. Although the Program String Structure shares some characteristics in common with several other programming systems, it adds a new dimension to the techniques previously available for effecting the linkage of arbitrary programs. The basic distinction between Program String Structures and all other techniques published to date is the use of the Buffer File mode of operation as the heart of the linkage mechanism.

The techniques advanced as part of Data Manager -1, or simply DM-1, come closest to affording the capabilities of the Program String Structure. Designed to satisfy general-purpose data processing requirements, DM-1 provides a structure for maintaining a library of programs and associated directories. The directories include a Program Description List and a Job Description List. An entry exists in the Program Description List for each program in the program library. The entry is comprised of the name of the program, the names by which inputs and outputs are internally referred to by the program and the descriptions of the formats of the inputs and outputs. An entry exists in the Job Description List for each job in the system. The entry is comprised of the name of job, the names of the programs which must be run to perform the job function, the inter-program output/input links and the free–standing inputs and outputs associated with the job.

Whereas DM-1 only requires an explicit specification of the format and structure of the data at the inputs and outputs of each program in the library, the Program String Structure additionally requires the assignment of symbolic names for such data format specifications. In addition to a Program Description Director and Job Description Director, the Program String Structure makes use of a Format Description Directory. An entry in the Format Description Directory exists for each different data format specification defined in the system. The entry is comprised of the name of the data format and the associated format description. Thus, a COMPOOL[3,4] type capability is provided which is not available in DM-1. During program assembly time references to data descriptions are incorporated into the body of the program. The programmer referencing data, such as in a table, for example, need not know the format of the data at the time the program is being designed.

The key difference, however, between the Program String Structures and DM-1 is the incorporation of the Buffer File mode of operation in the former case. One of the consequences of this is the absence of problems associated with synchronizing the concurrent execution of programs linked together in the DM-1 environment when using Program String Structures. Further, because of the difficulties associated with synchronizing the programs in the DM-1 environment, the programs to be linked are essentially run to completion before the data is passed onto another program. As noted in the previous section, the use of the buffer Files facilitates the passage of partial results from program to program.

Another technique which provides some of the capabilities afforded by the use of Program String Structures is expressed in the job control and task operation philosophies of Operating System/360[5] and its time sharing version, Model 67.[6] In these systems, the user is able to define jobs in terms of job steps, each of which can be a program previously entered into the program library. The binding of data to be operated on by a job can be delayed until the request is made to run the job. A job may also be stored away and later used as part of another larger job.

There is, however, a fundamental conceptual difference in the manner job steps are linked to form jobs under OS/360 and in the manner programs are linked to form larger programs as Program String Structures. Using the OS/360 approach, it is the programs which are considered to be in a moving stream, and control is passed from one program to another. Using the Program String Structure approach, program control remains stationary and the movement is of data from one program to another. The implications of this are related to the pre-planning involved in program preparation. In the former case, care must be taken when the program is designed to insure that control passes properly from one program to another. It remains the responsibility of the program to synchronize its operation with respect to other programs it may call on to be concurrently executed. In the latter case, the programs are independently prepared and the synchronization problems are handled automatically by the Buffer File linkage mechanism.

Use of the Program String Structure does not preclude taking advance of the recent thinking in some of the more advanced computer systems[6,7] with respect to when and how external symbolic references are resolved. For example, the notion of segment

linkages in the Multics-645/1 program system can be retained within the framework of the Program String Structure. The program, as used in the context of the Program String Structure, can be considered to be a set of segments whose inter-segment symbolic references are not translated to machine addresses until segment execution time. The linkage from program to program in the Program String Structure is superimposed upon this segment to segment linkage within a program. The design of a hardware implementation of the Program String Structure and Buffer File mode of operation in the Multics-645 environment is discussed in reference 8.

With respect to the Buffer File mode of operation, the authors are unaware of any efforts published to date which reflect either intentions or methods of implementation similar to those which have been described herein. Typical of the work which has been reported on is the STL integrated computer operating system described by Kory and Berning.[9] The system is comprised of several large computer processing units (IBM 7090's) which are linked to several small computer processing units (IBM 1401's) via a random access disk file. The 1401's handle the input/output from slower peripheral devices and place programs and data in the disk for execution by the 7090. Upon completion of the programs by the 7090, the results are placed back on the disk from where they are picked up and output by the 1401's.

The use of buffer areas described by Kory and Berning is only applicable for computer-to-computer communication and requires the use of a disk file outside of core storage as the buffer area. The use of Buffer Files as described in this paper, however, can be used for intra-computer as well as inter-computer program-to-program communication, and may make use of core storage for buffer areas. Further, whereas in the former case only data from completed jobs pass through the disk buffer area, in the latter case partial data from programs which continue to be executed are passed through the Buffer Files.

CONCLUSIONS

Program String Structures provide a mechanism by which programs which have been independently designed, coded, compiled, debugged and even modified can be maintained in a program pool and arbitrarily combined to perform user-oriented functions. The application of Program String Structure techniques completely eliminates the requirement for pre-planning on the part of the programmer with respect to how and with what data and other programs his program may some day be used.

The fundamental characteristics of the Program String Structure, which distinguishes it from any other program linkage technique reported on to date, is the Buffer File mode of operation. The Buffer Files provide a generic means for effecting inter-program communications whether the programs are all located in the same or in different computers. The Buffer Files automatically synchronize the operation of otherwise asynchronously operating programs, facilitate the passing of partial results from a program while it is still in progress to a second program, and simplify the passage of state and condition information between programs.

REFERENCES

1 E MORENOFF J M McLEAN
Job linkages and program strings
RADC TR 66 71 April 1966

2 J SABLE et al
Design of reliability central data management subsystem
RADC TR 65 189 July 1965

3 P DeSIMONE
SACCS master control
SDC FN LO 393 November 1960

4 B B MOORE
Program production system program description
SDC TM WD 141/000/01 January 1966

5 *IBM Operating System 360 concepts and facilities*
File No S360 36 Form C28 6535 0 IBM Corporation

6 *Systems 360 Model 67 Time Sharing System preliminary technical summary*
File No S/360/00 Form C20 1647 0 IBM Corporation

7 *GE 645 System Manual*
CPB 1231 unpublished advance information report
GE Computer Department

8 E MORENOFF J B McLEAN
Design of a Program linkage and communication mechanism for the GE 645 computer system
RADC TR 66 726 December 1966

9 M KORY P BERNING
The STL integrated computer operating system
Proceedings of the 19th National ACM Conference E2 1 1 E2 1 25 August 1964

DM-1–a generalized data management system

by P. J. DIXON and DR. J. SABLE
AUERBACH Corporation
Philadelphia, Pennsylvania

INTRODUCTION

I. Summary of Capability

The basic objective of Data Manager-1 (DM-1) is to provide business organizations with the comprehensive data handling capability outlined in this section.

It is assumed that the types of users will range from those who know nothing about the system and who wish to use it without learning more (e.g., line managers) to those who understand the system well and who wish to manipulate its inner workings to their advantage (e.g., programmers). Consequently, the system has a convenient user-oriented set of languages which shields some users from the complexities of the system.

The following system languages are designed to bring the user as close to DM-1 as possible:

(1) *Job Specification Language.* DM-1 is capable of storing both generalized system programs and specialized user programs in such a fashion that they can be called upon by a user to be run in the sequence he desires. This job specification capability is effected by means of a command language which is both easy to use and comprehensive in its ability to specify jobs.

(2) *Query Language.* DM-1 incorporates user-oriented query languages and is designed to incorporate other user-specified languages. It achieves this capability by utilizing an input interpreter which is driven by a syntax table.

Users can initiate conditional searches of arbitrary logical complexity by means of a dialog with DM-1, whereby the user makes a series of simple interrogations each building on the previous one. The products of the dialog culminate in a single complex query.

(3) *Data Definition Language.* Files or data sets of arbitrary logical complexity can be described and incorporated into DM-1 by means of an easy-to-use data definition language. The manual preparation of data descriptions for even complex data structures is reduced to an almost clerical operation, using elementary work-sheets.

The system does not specify or imply a data structure constrained by the nature of its design, but, rather, it provides a framework for building any reasonable logical data structure desired by the user.

(4) *Procedural Language.* Users can include their specialized task programs in the DM-1 repertoire by means of a procedural language. The current implementations of DM-1 employ JOVIAL, COBOL, and assembly languages.

All systems exist in an environment of change. The systems that adapt to such changes continue to give useful service. Several adaptive aspects of DM-1 are discussed below:

(1) *Modular Design.* Additions to the data base can be made once the definition of new data structure has been added to the system directory. New, specialized task programs can be added to the repertoire of the system with no new programming other than that of the task program itself.

(2) *Restructuring of Data.* In response to a user command, DM-1 is able to extract data from its existing logical structure and rearrange it into a structure more suitable to current use. Furthermore, the system retains statistical indicators of data usage so that it is able to respond to significant changes in usage patterns and to assist the data management personnel in changing the data structure.

In line with the objective of giving maximum convenience to the user, DM-1 is designed to communicate its results back to the user at the earliest possible moment, consistent with reasonable cost. The design objective of DM-1 is to provide the fastest response time available in time-shared multi-user systems by means of console access, where console equipment can be economically justified.

II. Functional Elements of DM-1

Data Manager-1 (DM-1) is designed to provide a

framework within which data may be analyzed, evaluated, summarized, and stored in such a way that it may be retrieved easily and quickly by users rather than data processing specialists. The system elements required to perform this function are the data base, the directories which point to the data, the system programs which use the directories to manipulate the data, and the Supervisor Program which processes the job request and controls the system.

Data base

Since the data base is the basic resource of DM-1, the fundamental strategy of the system is to retain the data in as flexible and accessible a form as possible. The DM-1 data description language not only permits the use of variable length fields and optional items but it also permits nested structures such as the nesting of a variable length file within a record of another file. A further feature is the ability to logically link items that are stored physically with other generically related items, so that a given item may be part of more than one parent item, in the sense of a lattice-type hierarchy. These links are kept in the DM-1 directory so that the identity of all items to be retrieved is known before data access is made.

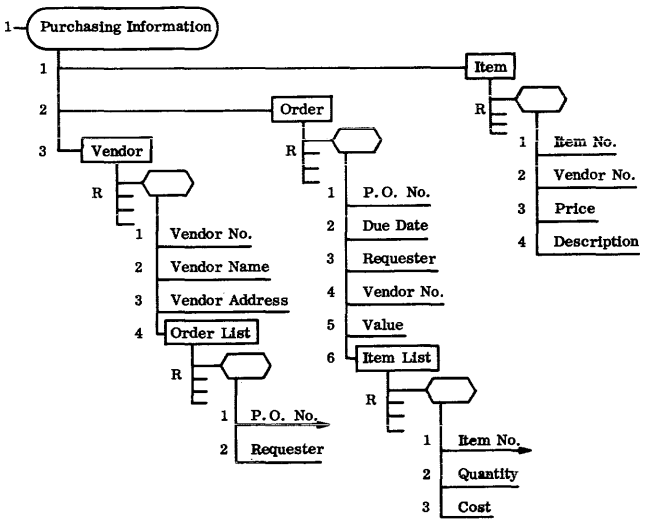
The data itself is either in random access or magnetic tape storage under the control of the operating system. Data is requested by means of a logical name which does not change with physical data movement. This permits the operating system to change the physical location of the data without having to notify DM-1.

The logical name of a data item is derived from the relative position of the item within the structure of the data base. A unique logical code is created for each item in the data base. The logical code is a numeric representation of the nodes in the tree structure of the data base and is called the Item Position Code. Figure 2-1 shows part of a hypothetical data base. The data items are indicated as branches of a central stem. Each branch is numbered in relation to other branches.

DM-1 supplies a standard language for defining the logical structure of data in the common data base. The language is supplied in two versions. The first uses an indented outline type structure on a standard form to signify the relationships between data items and subitems. The second uses formal parenthetic punctuation to signify data class relationships and is intended for those users who wish to use a linear parenthetic string input rather than a columnar page format.

Directories

DM-1 provides the user with a data description language which permits him to specify a very wide range of data structures. The structural description of the data



Note: The letter "R" indicates that the item is a record which is repeated in the data base as many times as the item is recorded.

Figure 2-1—Structure of purchasing data base

is implemented by means of the DM-1 directories which tie the system to the data base.

The data can be recalled to core memory only by means of system directories. A function of the directories is to translate the names of data items, first, into logical codes which describe the relative or logical positions of the items and, second, into the symbolic names of the data segment which can be used by the operating system to fetch the data. All system functions of file maintenance and retrieval depend on the directories to locate the desired data and to describe it once it has been found. The four main directories to the system are described in the following paragraphs.

Figure 2-2 shows what each directory requires as input and what each is designed to provide. The functions of the data directories are as follows:

(1) They are utilized to focus in on the data.
(2) They are utilized to give a description of the data in the record (e.g., whether the data is a floating point or an alphanumeric number).
(3) They are used to extract the data.
(4) They contain the index values for the data items so that searches can be performed within the directories without having to access the data.

The main directory tables are:

(1) *Term Encoding Table.* The basic function of the Term Encoding Table (TET) is to convert the name of a data item from its alphanumeric input form to a coded form which describes the logical position of the item in the data structure. The coded form, called an

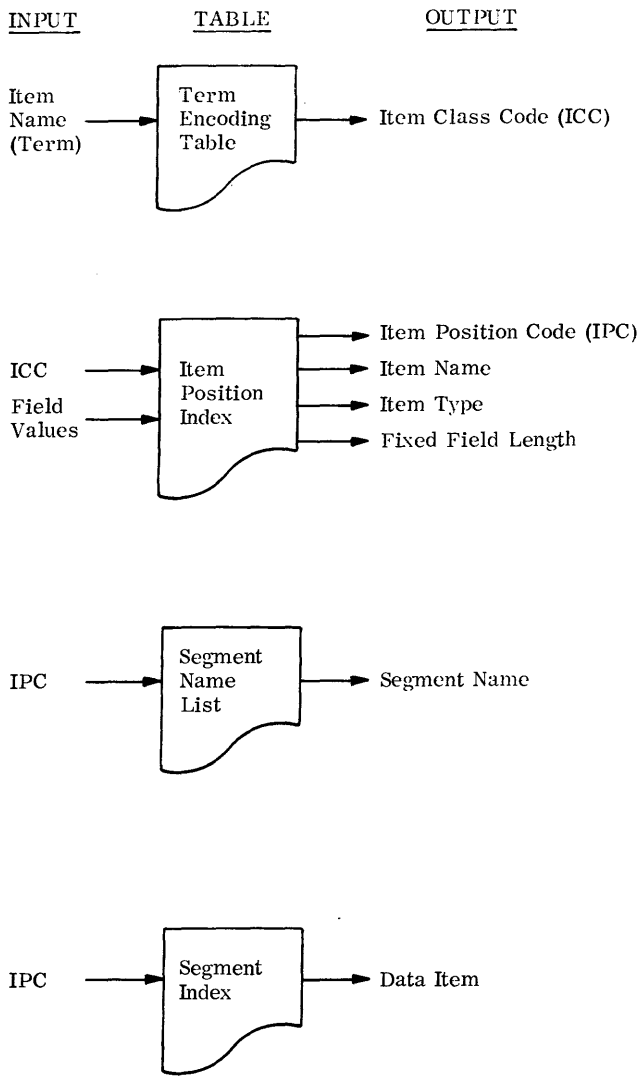


Figure 2-2—Directory inputs and outputs

Item Class Code (ICC), consists of integers which represent the nodes on the tree structure describing the data. For example, the ICC for Order List File in Figure 2-1 would be 1.3.R.4. A given term will have more than one ICC associated with it, if it is an item used in the data structure more than once.

(2) *Item Position Index.* The Item Class Code must be converted into a unique code by supplying values for the R's before a specific item can be retrieved from random-access storage. The conversion is performed by the Item Position Index, which is arranged by ICC and which contains all of the values by which certain data items have been indexed. When the ICC has been changed into a unique set of integers with no R values, it is called an Item Position Code (IPC).

Another function of the Item Position Index is to retain the statistical tallies of data usage, since all usage must pass through the Item Position Index. It is from these tallies that the need for data restructuring will become evident.

(3) *Segment Name List.* An Item Position Code is a code for a unique data item. It embodies sufficient information to be able to call for its appropriate data segment. Determining the name of the segment to be called is the task of the Segment Name List. When the name has been determined, it is given to the I/O Control Program along with a request for the segment, and the I/O Control Program retrieves the segment from peripheral storage and places it in a prescribed input area in core memory where the exact data item can be located.

(4) *Segment Index.* The final step in retrieving a data item is taken with the aid of the Segment Index. Each segment of data begins with a Segment Index which points to the data items contained in the segment. The Segment Index uses the IPC and auxiliary information from the Item Position Index to find its way to the desired item. At this point, there are several things which may be done with the data. The item may simply be retrieved, deleted from the segment, or used as a foundation upon which to add new data items to the file.

Job specification

A job is defined as a sequence of program tasks that accomplish some desired function for a user. DM-1 is designed to perform a variety of different jobs with a minimum of programming through the use of many generalized programs. The user will be able to call on these programs in the sequence he desires.

The user accomplishes this variety of uses by means of a Job Request. The Job Request performs the functions of item definition, data entry, program entry, job entry, and job running. The first four Job Request types perform system maintenance functions. The fifth Job Request type permits the user to run jobs of his own on the system.

A standard command language is provided for the user to specify each job he wishes accomplished.

Job specification resources afford both a dynamic and modular expansion to DM-1. New data can be added to the data base in batches or on-line, with few constraints on the type of data. Modular additions can be made to the system functions to incorporate new task-oriented programs. The Job Request types are summarized below:

(1) *Item Definition Request.* The logical structure of the data base can be altered through an Item Definition Request. An alteration is made in the directory entry which describes the relationship of the data item

to the data base. Possible alterations include the addition or deletion of data items such as statements, files, records, or fields.

(2) *Data Entry Request.* Data may be added or deleted by issuing a Data Entry Request. This type of request would normally be used for transactions where the data in the data base do not have to be examined in order to complete the transaction. More complex transactions may be effected by combining the Job Entry Request and Job Run Request which are described below. Data may be entered by means of a console or by magnetic tape. The data may be in the internal format of the system, or it may be in the external format of the data-description language.

(3) *Program Entry Request.* Descriptions of programs are entered into the system with the Program Entry Request. The programs and the descriptions of their parameters are given a unique name, and the descriptions are placed in the Program Description List where they are on call for the execution of jobs at a later time. Program Descriptions are used for checking purposes to be certain that subsequent jobs, which call on the programs, do so correctly, with the proper parameter specifications.

(4) *Job Entry Request.* The Job Entry Request is used to bring a Job Description into the system. A Job Description specifies the program to be run for a job and the sequence for running the programs. Any predetermined program parameters for the job may also be entered by this means. A Job Description is usually entered for a job which is run frequently, so that the job does not have to be specified each time it is run. A Job Entry Request causes a Job Description to be stored so that it may be called later by a Job Run Request. Upon entry, Job Descriptions are checked carefully by the system to be certain that they are compatible with the requirements of the programs they use.

(5) *Job Run Request.* A Job Run Request asks for a particular job to be run. A job must have been described previously in a Job Description by means of a Job Entry Request. If the Job Description specifies all parameters, the Job Run Request can be so brief as to simply identify the job name to be run. However, if the Job Description refers to several programs requiring parameters which have not been pre-assigned in the Job Description, the Job Run Request must specify these parameters in detail.

System programs

The programs in the system consist of two types:

(1) *System-Oriented Programs.* The programs are relatively stable and they provide the basic framework for processing data and for implementing other programs. They constitute the underlying programming foundation upon which all other system tasks are based. Examples of system-oriented programs are those which scan input data, access and manipulate system directories, locate data in the data base, and do routine jobs such as sorting and merging.

(2) *Task-Oriented Programs.* These programs are components of user jobs. They use the system-oriented programs to help them accomplish the required tasks. The task-oriented programs are less stable, in that they may be changed to suit the needs of an evolving task or the user easier than the system-oriented programs. That is, they may be added or deleted with much less effect on the system. Examples of task-oriented programs are those which summarize and analyze data and those which generate specialized reports.

The two types of programs are combined by a Job Description and work together to accomplish useful work. The data manager or systems analyst who is responsible for accomplishing a task creates a Job Description through a Job Entry Request. Although many system-oriented programs concern themselves with systems maintenance only, many others can be used as subroutines in user-oriented programs for sorting, merging, extracting, formatting, etc.

The use of the two program types may be illustrated by the following example. A technical specialist may wish to extract some prescribed fields of data from a permanent file in order to build a temporary file which is more suited to his purposes. He may plan to perform a series of analyses on the new file and have the results reported in a certain order and format. The specialist would issue a set of Job Entry Requests to produce the necessary Job Descriptions to tell the system what to do. The Job Descriptions would consist of a list naming both system-oriented and task-oriented programs. The system-oriented programs would scan the inputs which have been written in a user language, locate and extract data from the old file, construct the new file structure, and extract from the new file those data items desired by the analysis routines. If a generalized report generator were used, it too would be system-oriented, in that it would be available to all system users. The task-oriented programs would permit the system to select the data according to a Boolean logical condition, summarize the data, analyze it, and prepare the desired output if a specialized report routine were used.

System-oriented programs which deserve special mention are the Filesearch and the Table Access Packages. These generalized programs are responsible for all basic data manipulations and searches. They use the directories for extracting, storing, and altering data in the data base and for restructuring data into a

new form. One of the prime features of these programs is that they can make full use of the index data found in the Item Position Index to perform conditional searches of the data base. Further, they can manipulate the index data and reduce to a minimum the number of accesses to peripheral storage. This feature provides the foundation for accepting a query in any defined query language, and for processing it in an efficient manner.

III. Directories and the data base

A system—which efficiently processes data stored on random-access devices must rely heavily on directories to keep track of the data in the system. The directories, and the programs which utilize them, are critical elements of the system because they constitute the retrieval mechanism of the system and are used with great frequency during data manipulation.

The interaction between the directories and the data base is particularly important for two reasons:

(1) DM-1 directories are capable of being either specific or generic when they point to data. In contrast, the directories of many random-access data systems can be generic only, since they point to a relatively large group of data for retrieval and then scan it in core memory to determine which DM-1 part of it satisfies the search conditions. DM-1 can search in this generic way also, and since the DM-1 directories point to specific items, they can isolate small data items (e.g., fields or records of files imbedded in other files) which satisfy the search conditions before actually retrieving the data. This capability facilitates searching and increases the importance and utilization of the directories.

(2) DM-1 directories are used to describe the logical format of the data. Since DM-1 is sufficiently flexible to permit almost any kind of logical data format and structure, it must have a means of cataloging the various structures contained in the data base. The directories provide the "template" necessary to describe these structures, thus making the directories an indispensable part of the data handling process.

The following paragraphs describe the nature of the DM-1 directories by taking the reader through each functional step required to locate a particular item of data in the data base. This simple search process is only one of many uses for the directories, but it will serve to acquaint the reader with the basic mechanism for their use.

For the purpose of this description, it will be assumed that a user has entered a Job Request into the system to find a data record of a certain type. Although the case is artificial, let us assume that the user wants the first ORDER record in the ORDER file which represents a purchase order for J. Jones. The user knows there is a field in each ORDER record called REQUESTER, and he wants the system to search the ORDER file until it comes across a record with a value in the REQUESTER field which is J. JONES. The part of the Job Request that would describe this record (and others like it) might look like this.

ORDER, REQUESTER = J. JONES

The system would know that it must focus in on the item called ORDER and that it must select the first record which meets the criterion of having the value J. JONES in the field called REQUESTER. The following paragraphs describe how the selection is accomplished.

Term Encoding Table. The first step in locating the record is searching the Term Encoding Table for the Item Class Code (ICC) associated with the record. The prime parameter for searching the Term Encoding Table is a specific alphanumeric name of a data item called a term. If the term is unique in the system, it can be used alone. If the term is used to name items in more than one file, the term must be accompanied by terms higher in the data structure which give it sufficient context to make it unique. For example, Figure 2-1 presented a tree structure which included three occurrences of the term VENDOR No., in the records of the VENDOR, ORDER, and ITEM files. One must precede the term VENDOR No. by the term VENDOR, ORDER, or ITEM to make VENDOR No. refer to a unique item.

The Term Encoding Table is reached by means of the Term Encoding Table Directory. This Directory (see Figure 3-1) is brought into core memory, if it

ARGUMENT (Name of First Term in Segment)	FUNCTION (Segment Name)
ACCOUNTS	1862591
DUE DATE	7629720
PRICE	4790215
VALUE	2397528

Figure 3-1—Term encoding table directory

is not already there. The Directory contains an entry for each data segment of the Term Encoding Table. The argument of the Directory consists of the alphanumeric names of the first terms in each segment, arranged alphabetically. The function values of the entries are the names assigned to the segments by the I/O Control System.

The segment of the Term Encoding Table which contains the entry for REQUESTER is brought into core. The retrieval is performed by matching REQUESTER against the argument of the Directory.

For example, if the Term Encoding Table is four segments long, the Directory would have only four entries, as in Figure 3-1. REQUESTER is found to lie between PRICE and VALUE in the argument; hence, the segment of the Term Encoding Table which contains information about function starts with the term PRICE. The system might call this segment 4790215 (its symbolic name). Segment 4790215 is retrieved b ythe I/O Control Program.

The Term Encoding Table is almost as simple as its Directory but is much larger. The argument contains all the term names used in the system, arranged in alphabetical order. Figure 3-2 illustrates part of a

ARGUMENT (Term Name)	FUNCTION (Item Class Code- ICC)
COST	1.2.R.6.R.3
DESCRIPTION	1.1.R.4
DUE DATE	1.2.R.2
ITEM	1.1 1.1.R
ITEM LIST	1.2.R.6 1.2.R.6.R
ITEM No.	1.1.R.1 1.2.R.6.R.1
→ ORDER	1.2 1.2.R
ORDER LIST	1.3.R.R 1.3.R.4.R
PO No.	1.2.R.1 1.3.R.4.R.1
PRICE	1.1.R.3
PURCHASING INFORMATION	1
QUANTITY	1.2.R.6.R.2
→ REQUESTER	1.2.R.5 1.3.R.4.R.2
VALUE	1.2.R.3
VENDOR	1.3 1.3.R
VENDOR No.	1.1.R.2 1.3.R.1
VENDOR ADDRESS	1.2.R.4 1.3.R.3
VENDOR NAME	1.3.R.2

Figure 3-2—Term encoding table

hypothetical Term Encoding Table which includes most of the terms in the tree structure of Figure 2-1. The function consists of the appropriate Item Class Code for the term name. The ICC is derived by tracing the nodes in the tree structure to the item being coded. As mentioned previously, some terms will name more than one item within the tree structure, and a function entry will occur for each appearance of the name. For example, ITEM No. appears in both the ITEM File and the ITEM LIST File of the ORDER File and, hence, has two ICC entries in the function of the Term Encoding Table for the codes of the two item classes.

The Term REQUESTER is looked up in the argument of the Term Encoding Table. If only the ICC entry is found for REQUESTER, the item name is unique within the data structure, and the single ICC may be used. If more than one ICC entry is found, as in this case (see the bottom arrow in Figure 3-2), the desired context of the item must be established. The context is derived from the input term which precedes the term being processed. In this case, the term ORDER precedes the term REQUESTER in the input query. ORDER is found to have an ICC of 1.2 (see top arrow in Figure 3-2). The root of 1.2 is the proper context of REQUESTER, and 1.2.R.3 is selected as the correct ICC rather than 1.3.R.4.R.2. If the preceding term is insufficient to establish the proper context, the next preceding term is used until either the context is established or it is found that more input terms are required.

The primary output of the Term Encoding Table is a single ICC (in this case 1.2.R.3). Other outputs may consist of error messages if a term cannot be found in the Term Encoding Table, or if there are not enough terms to establish a unique context.

Item position index

The next step is to use the Item Position Index to convert the Item Class Code into an Item Position Code, thereby establishing a unique logical address for the data items. Two kinds of parameters are associated with the Item Position Index. The most important parameter is an Item Class Code. If the ICC represents a unique item instead of a class of items (i.e., there is no R in the ICC), no other parameters are required.

If the ICC contains the letter R, a further parameter must be supplied in order for the Item Position Index to convert the ICC into an IPC. The parameter may be a value for a field which has been indexed, such as in the present case where the value in the field REQUESTER has been designated as J. JONES. On the other hand, the parameter may be simply to obtain the first record in the file, or the next record in the file.

The Item Position Index is reached by means of the Item Position Index Directory in exactly the same fashion as the Term Encoding Table has been reached by the Term Encoding Table Directory. As seen in Figure 3-3, the argument of the Directory consists of ICC's arranged in alphanumeric order. The segment of the Item Position Index which contains information about 1.2.R.3. is found to start with 1.2, and the Segment Name is 4287933 (see the arrow in Figure 3-3).

The Item Position Index has an argument similar to its Directory: ICC's arranged alphanumerically. However, the argument of the Item Position Index contains *all* ICC's which are used in the system and is relatively complex, as seen by Figure 3-4. Note that this figure depicts only the logical relationships between portions of data in the directory but gives no indication of the

ARGUMENT (Name of First ICC in Segment)	FUNCTION (Segment Name)
0	5846932
1.1.3.R.2	8746651
1.2	4287933
1.3	7434658
1.3.R.2	4963225
1.3.R.4.R.1	8924863
2.2.1.R.6	7329821
2.3.2.R.2.R	2352178
2.4.1.R	6484390

Figure 3-3—Item position index directory

physical location of the data. For instance, the Index Value data and R Value data will be in subordinate directories of their own, linked to the main Item Position Index by addresses.

back to its original alphanumeric term. This capability is particularly important in man-machine communications where machine codes must be translated before being presented to human operators. In the actual design, the term *entry* is a link address back to its associated entry in the Term Encoding Table, rather than a repetition of the term name itself. This technique saves space while losing no efficiency.

(2) **Item Type.** An ICC can designate an item class of several types, and the code for the type is entered in this part of the function. The most generic item type is a *statement* (e.g., PURCHASING INFORMATION), where the type refers to no specific entity of data. Another type is a *file* (e.g., ORDER File; 1.2), which contains any number of records. A *record* is a type having exactly the same logical structure as other records in the same file (e.g., ORDER Record; 1.2.R). A record may contain other files or it may contain fields of three kinds. A *variable* field (e.g., REQUESTER; 1.2.R.3) is one whose length changes according to the length of the data in the field. A *fixed field* (e.g., DUE DATE; 1.2.R.2) is always the same length, and the length in characters

ARGUMENT Item Class Code (ICC)	FUNCTION				
	Term	Item Type	Index Values	Index Code	R Values
1.2	ORDER	FILE			
1.2.R	ORDER	RECORD			5, 10, 12.
1.2.R.1	PO No.	10 CHAR		ALL	
		VALUE	0-1300		1,(2),3, 7.
		VALUE	1301-5000		4, 6, 8, 9, 11.
1.2.R.2	DUE DATE	12 CHAR		SAME	
		VALUE	JAN 66		4, 6, 7, 11.
		VALUE	FEB 66		1,(2),3, 8, 9.
1.2.R.3	REQUESTER	VARIABLE		SAME	
		VALUE	A. SMITH		(2),4, 9, 11.
		VALUE	H. ALT		1, 3, 7, 8.
		VALUE	J. JONES		6.
1.2.R.4	VENDOR No.	10 CHAR		NONE	
1.2.R.5	VALUE	VARIABLE		NONE	
1.2.R.6	ITEM LIST	FILE			
1.2.R.6.R	ITEM LIST	RECORD			(Link to R_1 Values)
1.2.R.6.R.1	ITEM No.	6 CHAR		NONE	
1.2.R.6.R.2	QUANTITY	4 CHAR		NONE	
1.2.R.6.R.3	COST	10 CHAR		NONE	

Figure 3-4—Item position index

Figure 3-4 shows the logical relationships between entries in the Item Position Index. For each ICC in the argument, there are the following types of associated information:

(1) **Term.** The term *entry* after the ICC makes the Item Position Index the inverse table of the Term Encoding Table and permits the conversion of an ICC

is indicated in the function of the Item Position Index. The Item Type also specifies the mode in which the data of a field is stored, such as binary, octal, integer, decimal, floating point, or alphanumeric.

(3) **Index Values.** It is anticipated that many data fields will be indexed, thereby permitting searches for specific fields without having to access large amounts of data. When a field in a series of records is indexed, all the different field values in the records are listed under Index Values in the Item Position Index. If values are repeated, they need be listed only once. Reference can be made to the values without having to retrieve the records. An example in Figure 3-4 lists the field value range of zero to 1300 under the field called PO No.

(4) **Index Code.** Three indexing options are available to a program performing a file updating function. One option is to index all values which occur in a given field, whether similar values have been indexed or not (e.g., ALL, after PO No.; 1.2.R.1). Another option is to continue to index only those values which have already been indexed in the past (e.g., SAME, after DUE DATE; 1.2.R.2). The final option is to not index the field at all (e.g., NONE, after ITEM No.; 1.2.R.4.R.1).

(5) **R Values.** To give the indexing meaning, the index values must be directly related to specific fields. The relationship is provided by means of R Values which can be substituted for the letter R in the ICC. For example, Figure 3-4 shows that the term DUE DATE has an ICC of 1.2.R.2 and that it is an indexed fixed field. If one wanted to identify an item in this class which contained the value FEB 66, any of the R Values 1, 2, 3, 8 or 9 could be substituted in the ICC 1.2.R.2 for R to give a unique series of integers which would identify the exact relative (or logical) position of such a field in the data structure. The series of integers, with no letter R contained in it, is called an Item Position Code (IPC).

To demonstrate the use of the Item Position Index in locating specific fields of data, let us return to the example started at the beginning of this section. The ICC obtained from the Term Encoding Table is 1.2.R.3, and the field desired is called REQUESTER. The value of the field is specified by the Job Request as being equal to J. JONES. Assume that the portion of the Item Position Index given in Figure 3-4 is equal to a segment. The program searches the argument for the ICC equal to 1.2.R.3 and checks the term for consistency. The program notices that REQUESTER is a variable-length field which is indexed. It searches down the list of Index Values for J. JONES. When J. JONES is found, the R Value associated with J. JONES is extracted (in this case, the integer 6) and is substituted in the R of the record ICC to form the IPC 1.2.6. If more than one R Value exists, the other values can be used in subsequent searches for more transistor records having REQUESTER fields with the value J. JONES. The important fact to notice is that the IPC is now unique within the whole DM-1 data structure, and it may now be used to point to the exact data record desired by the user. (The IPC could have been used to point to a specific field or bit in the record.)

The problem of conducting Boolean searches for records meeting certain criteria can now be discussed with some understanding of the mechanism by which the searches may be conducted. Suppose that a Job Request wants all Order Records which have a purchase order number less than 1300, which are due in February, 1966, and which were requested by Mr. A. Smith. The program can examine the three fields, PO No., DUE DATE, and REQUESTER, for the desired Index Values. The list of R Values corresponding to these Index Values can be extracted and compared for any integers in common. In the case shown by Figure 3-4, the integer 2 is common to all three desired values. Hence, the R of ICC 1.2.R. can be replaced by 2, and the record identified by IPC 1.2.2 may be retrieved from random-access storage with full assurance that it meets the criteria of the Job Request Query. In this manner, the Item Position Index can be manipulated to determine what records are desired without having to fetch from storage any more records than those which exactly fit the criteria.

DM-1 files can be dynamic, with constant addition, deletion, and change. The Item Position Index is the logical place to keep track of how many records are contained in each file and of what gaps may exist in the logical structure of the file. Normally, an entry occurs in the R Value column of the Item Position Index only for indexed fields, but record fields are an exception. Figure 3-4 shows three integers in the R Values portion of the function for ORDER (1.2.R). The last integer (12) gives the R Value which should be used for the next record added to the file. Thus, the number of records in the file may be calculated by subtracting one from the last integer. The first (i.e., 5) and the second (i.e., 10) integers indicate R Values which were once part of the file but which have been deleted and not reused. New additions to the file can receive R Values for their IPC's by means of the entries in this column.

Two sets of statistics are retained in the Item Position Index to keep track of system usage. The first is a tally kept for each time a field value is used in a conditional search. Periodic analyses of these tallies permit the data base manager to unindex those values

which are seldom used, thereby conserving storage space. The second is a tally kept for each time a field is accessed, whether conditionally or as a result of serial processing. These tallies help to show the usage relationships between data items and aid the data base manager when he restructures data for more efficient processing.

When files are placed within the records of other files, multiple R's are found in the ICC's, and some complexities arise. For instance, a single set of integers is not sufficient to show the R Values for a single ICC, because the ICC now has multiple R's in it. Since each R represents a multiple number of records, the possible number of integer sets grows exponentially with each new R in the ICC. The problem of organizing the sets of R Value integers can be solved by linking with addresses each integer of the first set to its appropriate second level set, and so on down the hierarchy of R levels in the ICC.

There are several outputs which can be derived from the Item Position Index. The Item Position Code is the major output because it may be used to point directly to the data field being sought. The Item Type, another output, is also important because it describes the nature of the field being dealt with, such as the field length in the case of a fixed field.

A number of possible error conditions may be encountered while manipulating the Item Position Index: the desired IPC may not exist in the argument; the term in the Item Position Index may not agree with its counterpart in the Term Encoding Table; the field, for which a value was given on input, may be found to be unindexed; and many other possible internal inconsistencies. The output from the error conditions will depend entirely on the operating policy of the system at the time of implementation.

In this case, the output is an IPC with the value 1.2.6.

Segment name list

Looking up the name of the segment containing the desired data is the next step in locating the desired ORDER record. The names are kept in the Segment Name List. The parameter required to use the Segment Name List is a single IPC, which is used to request from the I/O Control Program the segment which contains the desired item.

As the data base grows, the Segment Name List may become large enough to require its own Directory. The Segment Name List Directory will be brought into core memory, if it is not already there. The argument of the Directory is a list of the first IPC in each segment of the Segment Name List (see Figure 3-5). The function is the name for each segment of the Segment Name List. The Segment Name List is brought into core memory, just as the Ter mEncoding Table was, selecting the segment with the next lower argument closest to the IPC being sought.

SEGMENT NAME LIST DIRECTORY

ARGUMENT (First IPC in Each Segment of the Segment Name List)	FUNCTION (Segment Name)
0	4968431
1.3.137.2	84268828
2.2.4.24.6	9206307

SEGMENT NAME LIST

ARGUMENT (First IPC in Each Segment of the Data Base)	FUNCTION (Segment Name)
0	6984320
1.1.2.5	7032694
1.2.4.1	8268215
1.2.11.4.19.3	4394762
1.3.7.4	1147238
1.3.84.1	6468945
1.3.112.6.1	2325464

Figure 3-5—Segment name list directory and segment name list

The Segment Name List is exactly like the Segment Name List Directory, only much larger (see Figure 3-5). Whereas the Directory argument has the first IPC of each Segment Name List segment, the Segment Name List argument has the first IPC of each data base segment. In essence, the Segment Name List is an extension of its own directory. Only directories can be ordered from the Local Control Program without using the Segment Name List, and this ordering is done through a directory which, in reality, is a small Segment Name List.

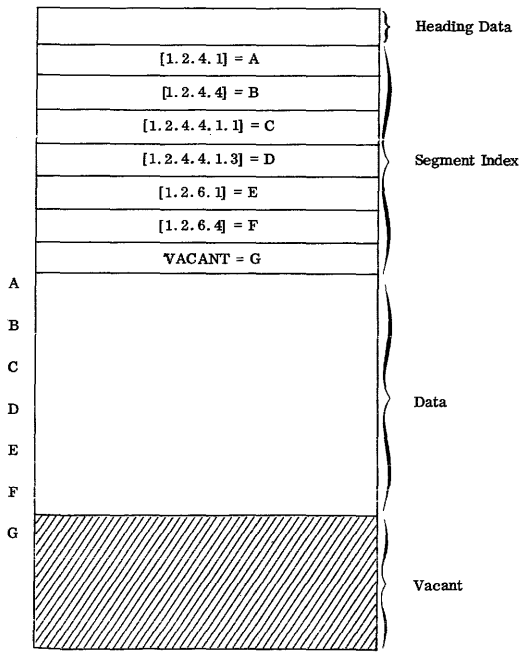
In our test case, the IPC desired is 1.2.6. The routine orders the first segment of the Segment Name List with the Segment Name of 4968431 (see the top arrow in Figure 3-5). The Segment Name List Segment brought into core memory looks something like the bottom table in Figure 3-5. The Segment Name List is used to select the data segment which contains IPC 1.2.6. In this case, the segment begins with IPC 1.2.4.1 and has a Segment Name of 8268215 (see the bottom arrow). The Segment Name is transmitted to the I/O Control Program, and an interpretive request is given the I/O Control Program to retrieve the data segment.

The Segment Name List also contains a Usage field to hold data on the number of times the segment has been retrieved in the last statistical data period. Analysis of this data will permit optimal use of the various storage media of the computer center.

Segment index

The final step in locating the data is made with the help of the Segment Index at the head of the segment containing the data. The major parameter for using the Segment Index is the IPC which is being sought. Another parameter is the Item Type (showing the length of the field, if it is a fixed field). A third parameter shows the beginning address of the segment in core memory. In actuality, the portion of the Item Position Index already referred to is another parameter, because the Segment Index and Item Position Index are used in conjunction with each other to locate the data item. A segment will be divided into four parts: a heading, a segment index, data, and vacant space. Except for the heading, the proportion of each part will vary, depending on how much data exist in the block and on how much indexing data are required to access the data.

To illustrate how the search would proceed, let us first consider what the string of data in the recently retrieved segment might look like. Figure 3-6 uses some symbols to show the various types of items and their logical relationship in the data structure. Although the symbols are shown on four lines, the items should be considered as one continuous string of data. The IPC for each item has been posted under the item, for visual reference. It will be noticed that the data string begins the segment and that the IPC value of 1.2.4.1 corresponds with the starting IPC of the segment just retrieved by means of the Segment Name List.



NOTE: A set of brackets represents an address of a location containing the data named by the IPC in the brackets. A capital letter represents a data address.

Figure 3-6—Graphic description of a data string

The DM-1 design specifies that the Segment Index consists of a list of addresses showing the starting position of data fields in the segment. Figure 3-7 shows a

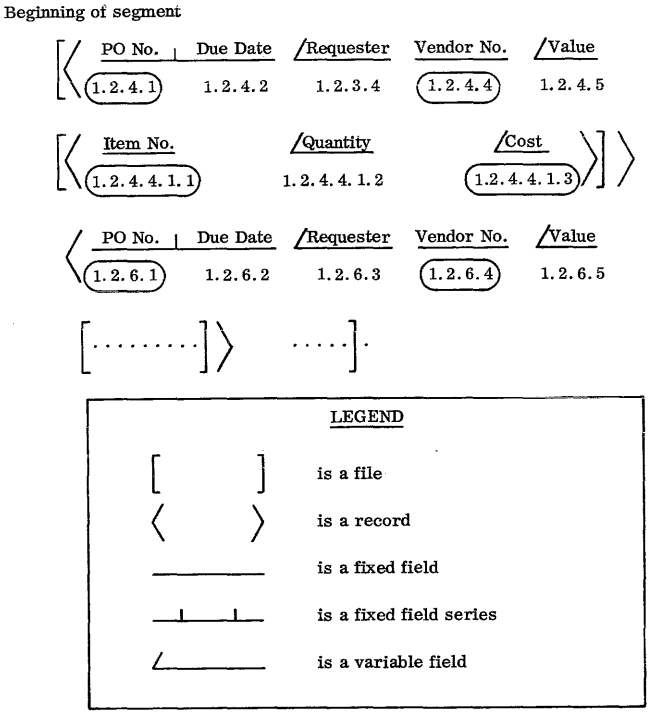


Figure 3-7—Sample segment index

sample Segment Index. The first address always shows the start of the data area (i.e., address A) and is relative to the beginning of the segment. The following addresses (i.e., B-F) point to subsequent data fields and are relative to the beginning of the data area. The final address in the Segment Index (i.e., address G) is always the end of the data area or the beginning of the vacant area. However, only those addresses that are vital to the process of locating data are included in the Segment Index. A series of fixed fields, for instance, needs only the starting address of the series, because the starting addresses of the rest of the fixed fields in the series can be calculated from the Item Position Index. Normally, only the end of variable or optional fields need be indicated. In Figure 3-6, the IPC addresses needed to scan the data have been circled. The other addresses need not be carried in any index, and it will be noticed that in Figure 3-7 only the circled addresses have been included in the Segment Index.

The basic logic for locating data in a segment is to use the Item Position Index to calculate where the desired address is in the Segment Index. The program starts with the ICC in the Item Position Index which corresponds to the IPC at the start of the segment. The program steps through the Item Position Index, while keeping count of how many addresses there should be

in the Segment Index up to that point. When it gets to the desired IPC, the program goes directly to the Segment Index and counts down through the index the same number of addresses to the desired address. This address points to the data with the specified IPC, thereby completing the retrieval function.

Normally, the program need go to the segment itself only once to fetch the data. If an optional field is discovered in the Item Position Index, however, the program must go to the data to see if the field actually exists.

To see how the process operates, let us follow the logic of how the data for IPC 1.2.6 is retrieved. The program knows from the Segment Name List Directory that the segment begins with IPC 1.2.4.1 and that the first address of the Segment Index will point to that data field. The program consults the Item Position Index for the Item Class Code which corresponds with 1.2.4.1. It is found to be 1.2.R.1. The R Values for the file ORDER show that the fourth record is a legitimate member of the file because no 4 appears under R Values to show that the fourth record is missing. The program notices that the first two fields are fixed, followed by a variable field, and the program deduces that a second address in the Segment Index will be required to point to the end of the variable field called REQUESTER or to the beginning of a fixed field called VENDOR No. Further, the program deduces that a third address will be required to point to the end of a variable field called VALUE or the beginning of a possible file called ITEM LIST. A link to a subsidiary list of second level R_1 Values (not shown in Figure 3-4) supplies the information that only one record exists in the ITEM LIST File when the first R in the ICC is equal to 4. The program deduces that a fourth address in the Segment Index must indicate the end of the QUANTITY field and the beginning of the COST field (IPC 1.2.4.4.1.3). A fifth address is needed to signal the end of the COST field, because it is variable, and the end of that inner file and record. It is also known that the fifth address skips to the beginning of record 1.2.6, because the Item Position Index R Values for ICC 1.2.R. state that record 1.2.5. does not exist in the file.

Since IPC 1.2.6 represents the desired record, the program can now turn to the Segment Index and select the fifth address in the index. The program can go to the data location in the segment which has that address, and the program will be positioned at the start of the desired record. The end of the record can be determined by finding the address which starts the next record, using the same procedure as described above. The intervening data may be retrieved, thereby completing the search process.

IV. DM-1 job requests

General

An important feature of DM-1 is its ability to respond to Job Requests, either immediately upon entry or after they have been stored in the system for a period of time. Much of the operational power of DM-1 is achieved by permitting users to sequence and link generalized programs by means of Job Requests. The remainder of this section covers a general concept of what happens within the system when each of the five job types is initiated.

Item definition request

An Item Definition Request can be used either to add or to delete the definition of items in the file structure. Changes are affected by means of a deletion followed by an addition which represents the correct version. The input to the system from an Item Definition Request which *adds* a definition will originate on a standard input form. The user will show the relationships between items and subitems by the use of indentations on the form, which may be tabbed. The item definitions will be keypunched to show the number of indentations for each entry.

Upon receipt of the definition data, the Supervisor program begins executing the program in the Add Item Definition job. The programs step through the terms of the input and note the logical relationships between terms. A series of records is created containing each term and its associated interim tree code, along with other information intended for inclusion in the Item Position Index, such as indexing code or Fixed-Field Length. The items in the records will automatically fall into order by tree code, because the input of the Item Definition Job Request would be in hierarchical order. The base node is checked to be sure it exists in the directory and that it can be used as a foundation for further expansion.

Separate records are prepared for each of the items. They are sorted by term name for the purpose of updating the Term Encoding Table. The original set of records is in ICC order, and it is used to update the Item Position Index by means of a single insertion, since the ICCs are in a cluster. A generalized print routine can be called upon to print out both series of records for manual accuracy checking and for updating manual records of the Term Encoding Table and the Item Position Index.

For *deletions* to the directories, a different process is employed. The last item in the input of the Job Request names the item to be deleted. In addition, all other items which are subsumed by the named item are to be deleted. The Term Encoding Table and the

Item Position Index are used to create the ICC of the item named for deletion. The entry in the Item Position Index is inspected for the item to be deleted and for all lower items in the same tree code branch (i.e., having the same ICC root). If the index shows that data exist in the data base for any of these items, an error condition is considered to exist because the definition for existing data must not be deleted. Data must not be permitted to exist in the data base without being properly defined. Thus, if data exist, the definition is not deleted, and a printout is prepared for the person initiating the Job Request instructing him to delete the data first if he wishes to delete the item definition.

If no corresponding data are found to be associated with the named item and its subsidiary items, the entries for the items are deleted from the Item Position Index. Pertinent items are deleted from the Term Encoding table, and a printed audit trail of deleted items is prepared.

Data entry request

The Data Entry Request may be used either to add data to or to delete data from the data base. With additions, the data are not *converted* from one logical structure to another, but they are *translated* from one physical structure to the IMS physical structure so that it can be incorporated into the data base. Data may be added in any one of four modes, as specified by the Data Entry Request (see the top half of Figure 4-1). The four modes are as follows:

(1) **Defined-External Mode.** This mode requires that an Item Definition, describing the data to be entered, has already been entered into the system and has been incorporated in the system directory. The data are entered in external format; that is, the data are manually formatted and named by a data specialist using a simple input worksheet.

(2) **Undefined-External Mode.** This mode has the same external format as the Defined-External Mode, but it does not contain a definition of the data being entered. Consequently, the definitions, in the form of an item image, must precede the data on the input medium.

(3) **Defined-Internal Mode.** This mode requires that a complete definition of the data be included in the system directory before the data are entered, just as in the Defined-External Mode. The format of the data, however, is different. The internal format is defined as being synonymous with the segment format used in the data segments of DM-1. Internally formatted data come in increments of a segment and have their own segment index at the beginning of each segment. The data are entirely dependent on the system directory for data identification and processing.

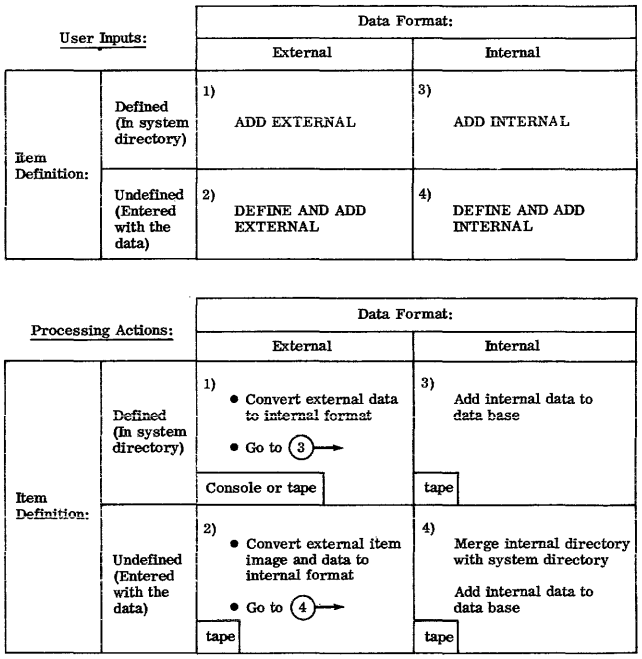


Figure 4-1—Data entry modes

(4) **Undefined-Internal Mode.** The last of the four modes does not require that the system contain the definition of the data to be entered, because, like the Undefined-External Mode, it contains its own definition. The data are in internal organization but the data also have a local Term Encoding Table and Item Position Index heading the blocks of data on the tape.

Program entry request

Compiled program code will be entered into the system according to established local conventions, and these programs will be called at execution time by means of the IOCS. Descriptions of the programs, however, are entered into the DM-1 directory for the purpose of inventory control and for internal checking to be certain that Job Descriptions are compatible with the programs they call on. The task of the Program Entry Request is to add or delete a Program Description from the Program Description List.

For an example of how a Program Entry Request might look, the reader is referred to the top portion of Figure 4-2. The example says, in English, "I want to enter two Program Descriptions into the system. My name is Jones and my identification number is 87. The deadline for entering the program is 9:30 AM on February 13. I wish to add to the Program Description List, and the Program Description will follow on the console. The first program I wish to enter is called ORDER SEARCH. The first input parameter of ORDER SEARCH is called REQUESTER, and REQUESTER must be an alphanumeric which is of variable length.

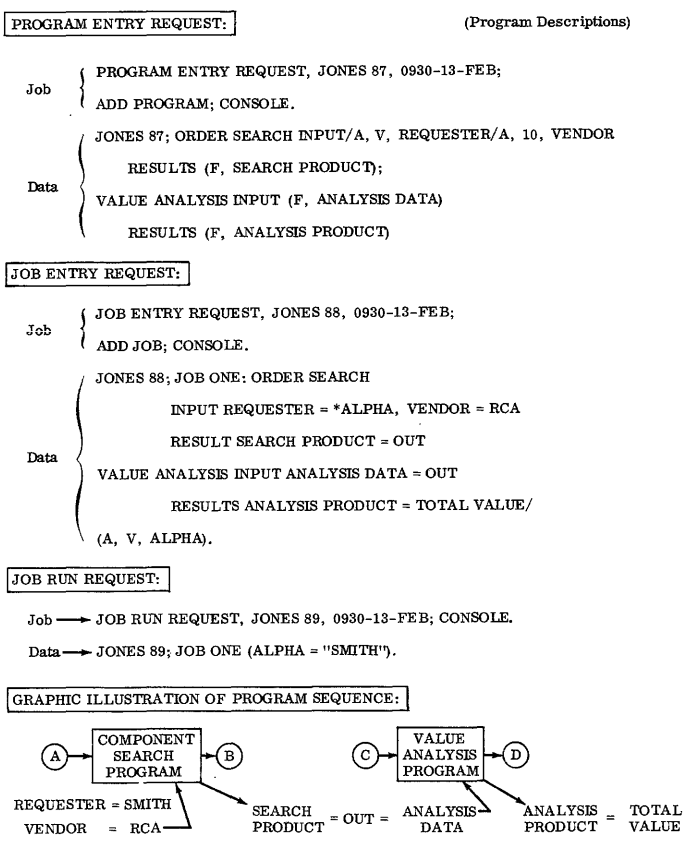


Figure 4-2—Sample job requests

The second input parameter of ORDER SEARCH is called VENDOR, and VENDOR must be an alphanumeric which is ten characters long. There is only one result (i.e., output) parameter for ORDER SEARCH, and it is a file called SEARCH PRODUCT. The second program I wish to enter is called VALUE ANALYSIS, and it has only one input parameter which is a file called ANALYSIS DATA. VALUE ANALYSIS has only one result parameter, and it is a file called ANALYSIS PRODUCT."

The basic functions of the Program Entry Request Routine may be summarized as follows:

(1) To associate a name with a program so that it may be called at a later time.

(2) To define the prerequisite parameters of a program so that the parameters given in any program description which calls for the program may be checked for format and completeness before the program is run.

Job entry request

The basic functions of the Job Entry Request are as follows:

(1) To check the consistency, accuracy, and completeness of an incoming Job Description by comparing it with the Program Descriptions to which the Job Description has referred.

(2) To merge a valid Job Description with the Job Description List so that it may be used at a later time for running a job.

The basic functions of a Job Description are as follows:

(1) To name the programs which must be run in order to accomplish the task assigned to the job.

(2) To specify the sequence in which the programs should be run.

(3) To specify the values or names required by the input parameters of the program to be used in the job, or to specify that the values or names will be given by the Job Run Request at a later time.

(4) To specify the names of portions of data resulting from the program so that these output parameters may be referred to and used by subsequent programs that have the same names specified in their input parameters.

(5) To specify the names of multiple program exits so that programs may be linked by the Job Manager of the Supervisor Routine according to various contingencies.

The Job Entry Request provides the raw data for a Job Description. An example of a Job Entry Request may be seen in Figure 4-2. Aside from the usual header information, the request says, "I would like to enter a Job Description into the system via the console." The job is called JOB ONE, and it calls for the running of two programs. The first program is called ORDER SEARCH. I do not wish to give a value at the present time to the first input parameter of ORDER SEARCH called REQUESTER. The value of TYPE will be named at execution time (when a Job Run Request calls for the job), so I will give it the job parameter name of ALPHA, and I will put an asterisk in front of the name to distinguish it from a value. I wish to assign the value RCA to the second input parameter called SEARCH PRODUCT. The second program is called VALUE ANALYSIS. I want to assign the name OUT to the input parameter, which the program VALUE ANALYSIS has called ANALYSIS DATA. (This assignment will connect it to the result parameter of ORDER SEARCH.) I want to assign the name TOTAL VALUE to the job result parameter, which the program VALUE ANALYSIS has called ANALYSIS PRODUCT. Since a value has not been supplied to the job input parameter called ALPHA (the program parameter called REQUESTER), I will place a description of the parameter requirements at the end of the Job Description for reference independent of the program. The requirement is that the parameter be a six-character alphanumeric.

The Job Entry Request processing requires that all

programs referred to by a Job Description be entered into the system before the Job Entry Request is processed. Processing takes the following form:

(1) The Program Descriptions associated with the Job Description are retrieved from the Program Description List. Any missing programs signal an error condition which terminates the routine. The Job Descriptions of both the Program ORDER SEARCH and the Program VALUE ANALYSIS are brought into core memory.

(2) The input and results parameters given in the Job Description are compared with their counterparts in the Program Descriptions to ensure that all parameters have been accounted for.

(3) At the same time, the limitations imposed on the parameters by the Program Descriptions are checked against the nature and format of the parameters given by the Job Description. For instance, the second parameter of the Program ORDER SEARCH, called VENDOR, must be an alphanumeric because the letter A is given. On checking the Job Description for the Program ORDER SEARCH, the routine finds the value RCA given. The value is an alphanumeric and satisfies the limitation. A number of the subroutines used to perform the above two tasks can be shared between the Program Entry Request Routine and the Job Entry Request Routine.

(4) One of the most important tasks of the routine is to scan the Job Description parameters for consistency to be certain that the result parameters specified match the input parameter requirements. For example, the result parameter, SEARCH PRODUCT, from Program ORDER SEARCH is called OUT. The routine scans for a matching input parameter which, in this case, is the parameter, ANALYSIS DATA, of the Program VALUE ANALYSIS. Once the match is determined, the limitations are compared. Any result or input parameters which require mates, but which do not have them, or which have improperly written mates, are printed out for subsequent correction.

(5) Any parameters which are unspecified at the time of Job Description entry are formatted at the end of the description for easy checking when they are specified by a Job Run Request.

Once the Job Description has been thoroughly checked and has been found to be valid, the routine calls in the proper segment of the Job Description List so that the new entry can be merged into the List, using the Job Name as the primary sort key. The entered Job Description is printed out for manual corroboration and for use as an audit trail.

The *deletion* of a Job Description from the Job Description List is a relatively simple task because no other portion of the system is functionally dependent upon it. Only a Job Run Request, as it is being introduced to the system, is dependent on a Job Description, and the Job Run Request is rejected immediately if the proper Job Description is not in the system.

Consequently, the Job Entry Request Routine can delete a Job Description by retrieving it from the Job Description List and restoring the List without the Job Description. The name of the job must be deleted from each Program Description that is referred to by the job.

File management on a small computer the C-10 system*

by GILBERT STEIL, JR.
The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

C-10 is a general purpose file management system that has been implemented by The MITRE Corporation for the IBM 1410. It is of interest for two reasons: first, it is one of the most flexible file management systems ever attempted for a machine the size of the IBM 1410; and second, the conditions surrounding its implementation have produced a design for a large system of programs that displays some uncommon and useful characteristics. These uncommon characteristics include a strong affinity for redesign; a framework for debugging that eliminates the need for memory dumps, symbolic snapshots, and other conventional debugging aids; and an adaptability to machines of various memory sizes and configurations without reprogramming.

Our purpose in presenting this paper, therefore, is twofold: to describe the C-10 file management system, and to relate the experience of its unusual implementation.

Evolution

Work on the C-10 system was started in December of 1964, and was originally slated to terminate six months later in June of 1965. "C-10" originally stood for "COLINGO-10" and was intended to be an improved version of a small general purpose file management system named "COLINGO" which MITRE had developed earlier. In return for a slightly larger machine (the IBM 1410 instead of the IBM 1401, on which COLINGO operates), some increased flexibility was expected. In particular, the design goals for C-10 included the following capabilities that COLINGO did not provide: cross-file referencing, multi-level files (files in which property values may be filed), an improved means of generating new files, more elaborate facilities for generating formatted reports, and increased speed for complex operations. An important restriction at that time, which must be noted now, was that the work was to be completed in no more than six months.

The designers examined the problems of achieving these improvements within the COLINGO software framework with the available manpower in the available time, and decided it was impossible. Several other approaches to implementing an improved COLINGO on the IBM 1410 were then investigated. The adopted approach was to start from scratch and build a LISP-like interpreter defined over a single data structure, with the intent of programming 90 percent of the system in the language of the interpreter. This approach, while thought to be a long shot, was made inviting by what appeared to be an ideal data structure both for general system use and as a basis for building files. This data structure is now known as "streams" (see Appendix II). The stream structure was thought to be suitable for modeling files because it is possible to take advantage of a specialized instruction of the IBM 1410 (LLE: Lookup on Less than or Equal) to implement an important primitive operation in one instruction.

System analysis and design took place in two weeks. A work schedule was drawn up, and work was begun. For the first six to eight weeks the schedule was rigorously maintained. After a few weeks, the IOCS module was operating, slightly later the disk allocator was debugged, and soon the relocatable subroutine controller was working. In the meantime, as a result of other work, the bulk of the system had been coded in the LISP-like interpretive language. Then, about three months into the project, a large snag was encountered. At this time the LISP-like interpreter became operational and some of the interpretive pro-

*The material contained in this article has been approved for public dissemination under Contract AF19(628)-5165.

grams were checked out. Debugging the logical flow of the interpretive programs went smoothly enough, and if the only concern had been the logic of these programs, the tight schedule could have been met. But performance was also important, and unfortunately the performance of these programs was off by not just a mere factor of two or three, but by *several orders of magnitude*. For example, a subroutine in one of the language translators consumed not an intended 10 milliseconds of time, but 3 or 4 *seconds*!

Soon it was realized that disruptions to the work schedule due to the performance problem would not allow a product to be delivered on time. But as hopes for meeting the schedule were dimmed, considerable interest in surmounting the performance problem, and in the system design in general, was generated. As a result, the C-10 project evolved into an experiment in system design and modification which continued not only for the remainder of the originally scheduled six months, but for an additional fourteen months afterward. During this additional time the system underwent many experiments in restructuring, which included: redesigning and recoding important primitive operations; replacing one data structure with another; writing a compiler for translating the LISP-like language into machine language; replacing LISP-like programs with machine-language equivalents; and replacing machine language programs with LISP-like equivalents. At the end of this period the performance of the system had been brought up to a satisfactory level, and the system had evolved into a flexible environment for file management based on several data structures and written in several languages.

At the present time no further development work is contemplated, but the system is being maintained and used experimentally on both the IBM 1410 and the IBM 360/40 (under emulation).

File management

General purpose file management systems are characteristically directed toward a user with routine data processing problems involving large files of information. They are designed on the basis of a generalized file structure, and consist of a set of generalized file management functions, embedded within an operating system. These generalized file management functions usually include:

(a) generation of structured files from any machinable data
(b) maintenance and updating
(c) retrieval of selected data
(d) generating formatted reports
(e) operations involving combinations of the above

Thus, it is intended that users of the system need not program their problems from scratch, but need only supply an appropriate set of parameters to the already existing generalized tools. Several general purpose file management systems have been implemented for machines of various sizes, or are now in the implementation process. These include ADAM[1] on the IBM 7030, LUCID[2] on the AN/FSQ-32V, and GIS[3] on the IBM 360.

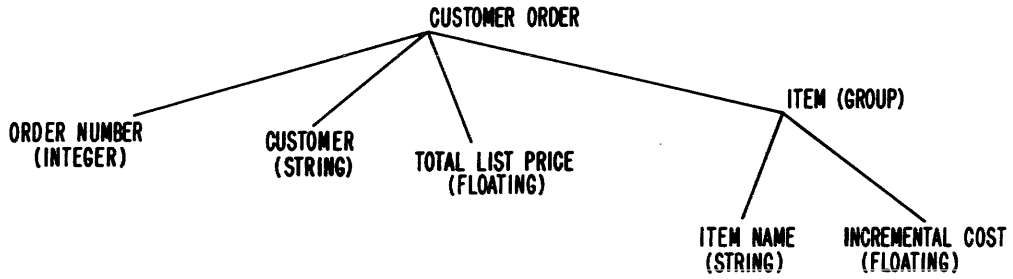
The C-10 generalized file structure can be described as follows: A file is a linear sequence of *objects*. Associated with a file is a set of *properties* common to all objects. Each object contains a set of *property values*. For example, a CUSTOMER ORDER file might have associated with it the set of properties, ORDER NUMBER, CUSTOMER, TOTAL LIST PRICE, and ITEM. Each object of the file contains a property value for each of the properties; for example, 3246, SAMPSON, 3018.30, etc. (See Figure 1).

Each property has a value *type*, which remains the same throughout the file. The type of a property may be *integer, string, floating, group,* etc. If a property has type *group*, the property values for that property are themselves entire files (see, for example, ITEM, in Figure 1). In this manner, files can be nested within files to any level. This structure is sometimes referred to as an "N-level file structure." Properties may have additional attributes, such as length for strings, formats for printing, etc. All C-10 files reside on disk, although it is possible to save them on tape.

Most file management systems are organized into modules or phases along the lines suggested earlier, that is, into generation, updating, retrieval, and so forth. This frequently leads to inefficient processing in the case that a task is not a simple generation, update, or retrieval, but is some combination of these operations. This is because each separate operation involves at least one access to a file and sometimes an entire pass through a file. In some systems, separate languages are even used to describe the separate processes. In summary, the traditional approach to file management systems is to pass the *data base* over each *procedural module* as many times as are necessary to achieve the desired operation.

C-10 file processing is a departure from this tradition. In C-10 it is possible to pass the entire *procedure base* over each *data module*. Here is how this works: In C-10 a single language, PROFILE, is used to express all file management functions. The PROFILE user can manipulate as many files as he wishes simul-

STRUCTURE DESCRIPTION OF THE CUSTOMER ORDER FILE:



TABULAR DESCRIPTION OF THE CUSTOMER ORDER FILE, WITH VALUES GIVEN:

3246	SAMPSON	3018.30		
			2 DR. SEDAN	2680.00
			AUTOMATIC	281.50
			RADIO	56.80
3251	FLINGER	3042.45	2 DR. COUPE	2640.00
			4 SPEED TRANSMISSION	290.80
			DISC BRAKES	81.90
			HEAVY DUTY SUSPENSION	13.80
			FAST MANUAL STEERING	12.70
			HEAVY DUTY STOCKS	3.25

Figure 1 – Example of a C-10 file

taneously. The basic mechanism that he has available to him is a position operation that selects an object of a file for processing. Once an object is selected, its property values may be extracted, changed, or deleted, or an additional object may be inserted after it. Files may be positioned forward or backward, or directly to a specific object that is pointed to from an index (which is itself a file). Data can be transferred directly between files. The same positioning mechanism is also used within groups (subfiles). Data can be transferred from any level to any level.

In a similar way, the PROFILE user directs data from and to I/O devices. PROFILE allows the user to manipulate data from (to) any input (output) device as though it were a continuous stream of characters, indexed by a pointer. The pointer may be moved forward and backward, and data are extracted from (entered into) the stream relative to the position of the pointer. Thus, it is possible to write PROFILE procedures that are quite flexible; for example, a procedure may generate a new file from data contained in several old files as well as data supplied on cards. The C-10 file management environment as viewed from PROFILE is illustrated in Figure 2. Here we see several streams of data flowing from input devices and referenced with a pointer, several files with particular objects selected for processing, several streams of data flowing to output devices, a collection of other data structures, a set of variables, and a library of procedures.

File management problems are solved in PROFILE by segmenting the task into several procedures and writing and debugging each procedure separately. Procedures are maintained in a library, and a procedure may appeal to any procedure in the library (including itself) to have a subtask performed.

Finally, it should be pointed out that PROFILE contains a set of statements specifically intended for spontaneous on-line work. Of course, any PROFILE procedure can be written, debugged, and added to the procedure base on-line.

A detailed example of the use of PROFILE is illustrated in Appendix I.

Implementation

In the previous section we discussed how C-10 looks to a user involved in file management applications. In this section we will discuss how C-10 looks to a C-10 systems programmer. Since there is no

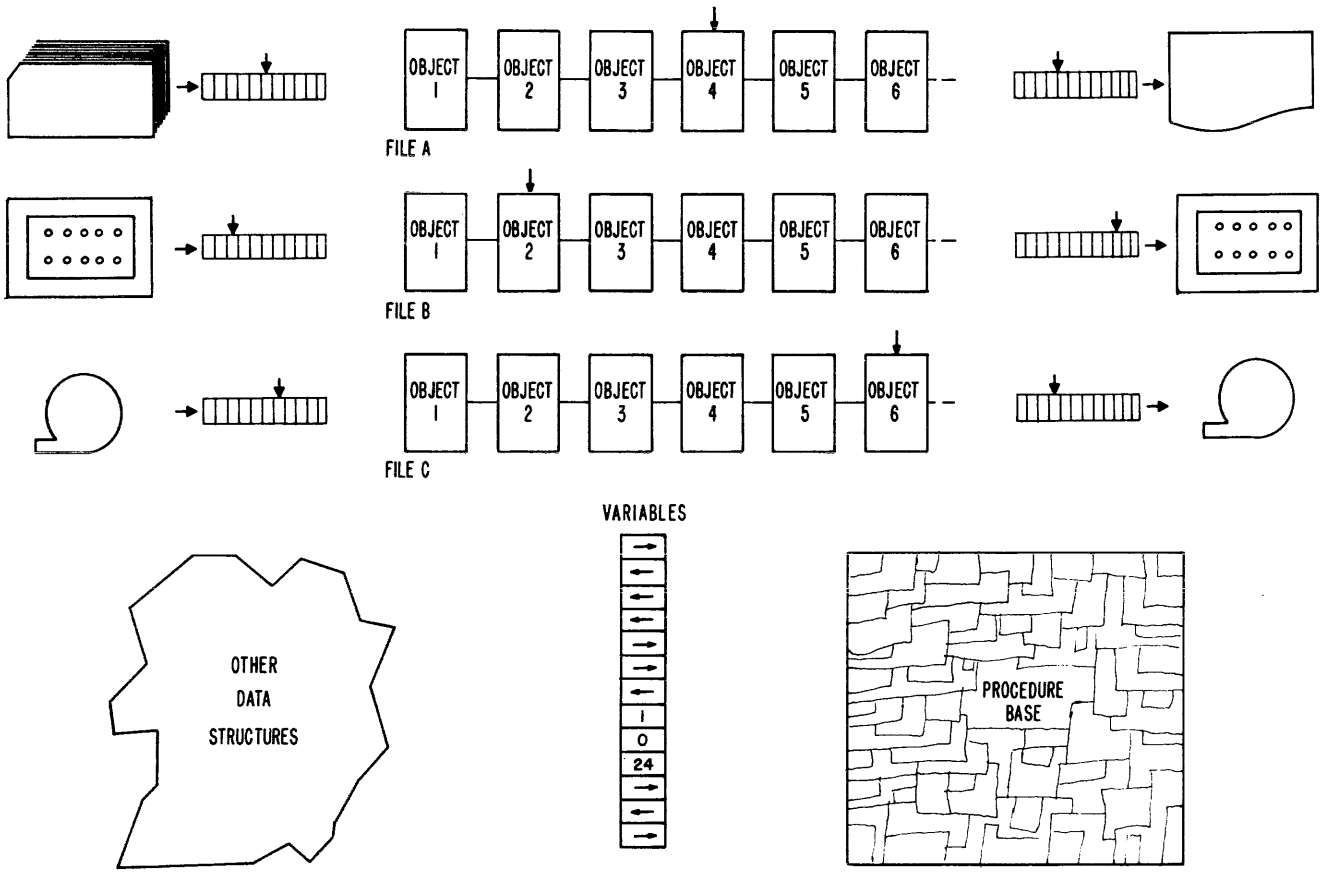


Figure 2—The C-10 file management environment as viewed in PROFILE

distinction made between user procedures and system procedures,* and therefore no distinction between users and systems programmers, we are really discussing how C-10 looks to a sophisticated user.

To the systems programmer, the C-10 system offers a framework within which he can perform his daily work. It allows him to define the pieces of his program one procedure at a time, and it allows him to save the state of the entire system from one session to the next. It provides him with the following important tools.

The system provides an expandable set of data structures, which so far includes files, private stacks,† streams,† and blocks.† All of these data structures have the characteristic that they are virtually infinite in length, and all storage management problems associated with space sharing of core and disk are handled implicitly. This means that the systems programmer can create and use instances of any of these data structures without consideration of a proliferating maze of details and qualifications. It is also important to note that instances of any of these data structures are not tied to a specific procedure, as is true in most programming languages. An instance of a data structure may be created by one procedure, saved by a system mechanism, and referenced later by a second, independent procedure. Each of the system data structures is defined over a set of data types which are a subset of the data types known to the system. The set of system data types is also expandable.

The system provides an expandable set of source languages, which so far includes AUTOCODER, STEP, and PROFILE. AUTOCODER is the assembly language of the IBM 1410; STEP is a language similar to LISP (but defined over streams instead of lists); PROFILE is the file manipulation language described above. A matrix showing which data structures may be referenced in each of the languages is shown in Figure 3. The system provides a generalized macro facility, TAP, which may be used with any source language except AUTOCODER.

The system provides an expandable set of object languages, which so far includes machine language

†Defined in Appendix II.

*Except when it comes to maintaining procedure libraries for separate users.

DATA STRUCTURES / DATA TYPE	FILES	PRIVATE STACKS	STREAMS	BLOCKS
INTEGER	✓	✓	✓	
FLOATING POINT	✓	✓	✓	
STRING	✓	✓	✓	
POINTER	✓	✓	✓	
BULK				✓
BOOLEAN		✓		

SOURCE LANGUAGES / DATA STRUCTURES	AUTOCODER	STEP	PROFILE
FILES	—	—	✓
PRIVATE STACKS	✓	✓	✓
STREAMS	✓	✓	✓
BLOCKS	✓	✓	✓

SOURCE LANGUAGES / OBJECT LANGUAGES	AUTOCODER	STEP	PROFILE
MACHINE LANGUAGE	✓	✓	✓
STEP INTERNAL		✓	✓

Figure 3 – Relationships Among the Tools Provided by C-10 For the Systems Programmer

and STEP internal; and, of course, the system provides a set of miscellaneous translators for translating source language procedures into object language procedures. A procedure in either object language may appeal to any procedure in the other object language. Figure 3 also includes matrices showing which source language can be translated into each of the object languages.

One thing the system does not automatically provide is an executive routine. This is because each user tends to use the system in a different way, and it was not considered reasonable to build a large monolithic executive which anticipated the requirements of each user. What is provided instead is a set of tools that permit the user to piece together his own executive routine. All the basic modules of the system – an editor, translators, compilers, interpreters, allocators – have simple interfaces and can be combined in an arbitrary way.

The system provides a set of debugging tools, the mainstay of which is a dynamic, selective trace. This trace interprets and prints the arguments and values returned by each selected procedure. It is effective because C-10 programs consist of a large number of small procedures. Memory dumps, patches, and symbolic snapshots are almost never used. All communication between procedures is accomplished through an intermediate linkage routine, and it is here that the tracing mechanism is housed. The intermediate linkage routine would also be an ideal place to house instrumentation concerned with timing, but unfortunately an adequate clock is not available on the IBM 1410.

The C-10 system has been designed to adapt readily to minor changes in the machine configuration. When initially loaded from an arbitrary tape unit, it holds a dialog with the operator to determine the current machine configuration. It will operate with either a 1301, 1302, or 1311 disk unit (2311 or 2302 on IBM 360).

Additional core storage does not necessitate any reprogramming, and system operation speeds up as core is added. C-10 operates a little more than twice as fast with an 80,000 character memory as with a 40,000 character memory.

Finally, the system allows the user access not only to the large modules of the system (like the STEP compiler, for example) but also to each of the individual procedures that constitute the large modules.

Every procedure in the system has a simple interface, and is documented as though it were independent of the rest. This is an advantage that we feel is important, and deserves more explanation.

Imagine that you have the job of writing a compiler for a computer that comes from the manufacturer equipped with a conventional operating system, and ALGOL and COBOL compilers. In the construction of your new compiler you will need subroutines that manipulate character strings, subroutines that format error messages, subroutines that prepare output, etc. Doubtless, subroutines that perform these jobs already exist in the software supplied with the machine. But do you have access to them? Definitely not! Even if they were separately documented and maintained, it is unlikely that all the boundary conditions surrounding their use could be satisfied.

Things are very different in C-10. Every module consists of a large number of small procedures, and each procedure is independent, with a necessarily simple interface that is documented. As a result, there is a great deal of sharing between modules that perform similar functions. Because of the extensive use of this feature, the C-10 system is relatively small. The entire system—two compilers, one interpreter, an editor, disk and core allocators, data structure primitives, etc.—is composed of approximately 400 procedures, averaging 900 characters in length.

Commentary

The two most important factors responsible for the uncommon characteristics of the C-10 system are the size of the machine on which it was implemented (hence the title of this paper), and the unusual conditions surrounding the development of the system (hence the discussion of system evolution). The small core size (40,000 characters) forced a design which (1) assumes that things seldom fit into core, and (2) provides an environment that allows the systems programmer to forget about the storage allocation problem (unless, of course, he is writing an allocator). The magnitude of the task, and the short time in which it originally was to be accomplished, motivated a design that attempted to trade off operating efficiency for speed of construction and ease of debugging. This trade-off succeeded too well, with the loss, initially, of far more performance than could be afforded!

One of C-10's important characteristics is an affinity for redesign unknown in our experience. During the course of improving the system every module was programmed at least twice; more efficient data structures were invented and the old ones were replaced with new ones; tools were added that helped evaluate system performance. In one case, the representation of files was changed drastically, and this was done without severe interruption to the day-to-day use of the system.

Other ways in which this system differs from others within the scope of our experience are that it is significantly easier to document, it is easier to debug, and it exhibits a good economy of programs. All of this we trace to the *enforced* uniformity and simplicity of procedure interfaces, the nearly exclusive use of data structures of virtually infinite length, and the emphasis on small, independent procedures. Perhaps C-10's greatest contribution to program construction is its service as a discipline.

In the long run, the attempted trade-off of operating efficiency for speed of construction failed. By the time system performance was brought up to a satisfactory level by redesign and tuning, the system could have been constructed along more conventional lines. The trade-off that was, in fact, accomplished in the final product is one of operating efficiency in exchange for an affinity for redesign, ease of documentation, ease of debugging, and economy of programs. Now it occurs to us that for the construction of very large programs in an environment where the cost of manpower is increasing while the cost of hardware is decreasing, trade-offs of this type are very much desired.

The construction of a file management system like C-10 surely should be, essentially, the construction of an application program. Yet, looking back, it appears that only a small percentage of the total effort was concerned explicitly with the problems of file management. The remainder was spent solving *programming problems*, that is, building the programming environment that we have described above. This circumstance is not unique, and seems to come up with the construction of almost every medium-sized or larger program. See, for example, the discussion of the implementation of ALPAK, and the subsequent design of the programming environment, OEDIPUS, in Reference 4.

The development of the PROFILE language was an evolutionary process. Starting as a highly restrictive "near-English" query language, it was gradually generalized to the point where it combines file generation, retrieval, updating, and report formatting in one language. In its present form it is a programming language, although a simple one. The fact that it is a programming language is met at first with some grumbling on behalf of its users, but they find it easy to learn and they are usually pleased with the flexibility it has over "near-English" languages.

If C-10 were to be designed again, two mistakes would certainly be corrected. The first concerns what we have come to call "system arrogance." Although any procedure within C-10 can be appealed to by a procedure within the system, the system itself cannot be used as a subroutine of another system. In this way the system is "arrogant." The second mistake was in not building an assembler as an integral part of the system. Instead, all assemblies are made off-line with an "arrogant" assembler that was found impossible to incorporate into C-10. The decision to use AUTOCODER was justified by the original schedule, but it has led to some unfortunate results. In particular, the system cannot be easily used to maintain its own symbolic decks, and editing and macro facilities which are otherwise general purpose cannot be used on AUTOCODER source statements.

Finally, one thing that was done correctly was the strict adherence to configuration independence. When operating the system at other installations, it is a valuable asset to have the system automatically adapt to any available disk unit and to take advantage of increased core storage space.

CONCLUSIONS

C-10 has been described as a file management system on a small computer. It is unusually flexible in that it allows any number of files to be processed concurrently, it allows file processing operations to be combined with file generation and report generation, and it enables and encourages its users to break their problems into small procedures.

C-10 has been described as an environment for programming that provides facilities not usually found in an operating system, especially for a machine the size of the IBM 1410. These facilities include:

(a) A framework that allows the programmer to define and check out one procedure at a time, within the system environment, and to save the state of the system from one session to the next.

(b) An expandable set of data structures characterized by virtually infinite length.

(c) A choice of three languages in which to express procedures, and the ability to appeal to any procedure from any procedure.

(d) A set of debugging tools, which are made effective by a reliance on small procedures.

(e) Access to the subroutines of a module that is already part of the system.

(f) The capability of adjusting to changes in configuration without reprogramming.

The most important characteristic of C-10 as a programming environment is that it is a *nucleus* of an integrated software system that can be readily changed and molded to fit the needs of a specific set of users. It is *not* a large, complex, monolithic set of programs which attempts to be all things to all men.

REFERENCE

1 T L CONNORS
ADAM - a generalized data management system
Proceedings AFIPS Spring Joint Computer Conference
Spartan Book Co Washington D C 1966

2 E E GRANT
The lucid users' manual
SDC TM-2354/001/00, System Development Corporation
Santa Monica California June 16 1965

3 J H Bryant P SEMPLE JR
GIS and file management
Proceedings
21st National Conference of the ACM
Thompson Book Co Washington D C 1966

4 W S BROWN
An operating environment for dynamic-recursive computer programming systems
Communications of the ACM
Volume 8 number 6 pp 371-377 June 1965

5 C BAUM L GORSUCH eds
Proceedings of the second symposium on computer centered data base systems
SDC TM-2624/100/00 System Development Corporation
Santa Monica California

6 L A BACHOROWSKI
Phase II demonstration
MTR-93 Supplement 2
The MITRE Corporation ESD-TR 66-647
Bedford Massachusetts July 5 1966

7 G STEIL JR
C-10 user's manual
MTR-35
The MITRE Corporation Bedford Massachusetts
ESD-TR 66-335 March 15 1966

APPENDIX I
A file management example

The Second Symposium on Computer-Centered Data Base Systems[5] was held on September 20-21, 1965. Prior to the symposium, a data base problem was submitted to the designers of five file management systems. The problem was divided into seven parts, and C-10 solutions to all seven parts are presented in Reference 6. In this Appendix we summarize the C-10 solution to two of the seven parts: the production of two periodic reports, and the spontaneous generation of a demand report.

The periodic report problem concerns two files: the PERSONNEL file and the ORGANIZATION file (see Figure 4). Units within the organization are either divisions, departments, groups, or sections.

The periodic report request is stated as follows: "Combine the Personnel information with the Organization information, so as to prepare two reports. One report covers all sections in ascending organization unit number sequence, showing for each section its authorized complement, actual complement and deviation from budget. The second report presents this same type of information for all groups."

The procedure, as illustrated, has two arguments. The first argument designates the organizational level for the report, that is, section, group, department, or division. The second argument, a number, limits the number of units to be processed, and was included solely to make it possible to control processing time during demonstrations.

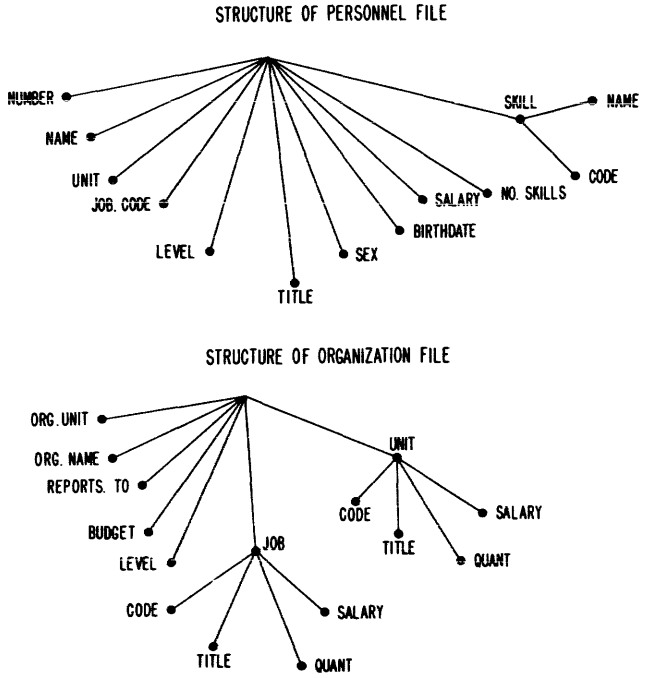


Figure 4—Structure of the Personnel and Organization Files

This problem is not the most difficult of the problems posed at the symposium. It does, however, involve cross-file referencing, calculation, and the outputting of reports in a specific format. The C-10 solution makes only one pass of each file for a report, and is illustrated in Figure 5. Comments in the actual program will help the interested reader get through it. Sample output is shown in Figure 6.

Generating the demand report is simple, and can be accomplished easily on-line. The problem asks that the name, number, unit, code, and skill names be retrieved of all personnel satisfying the following criteria: more than two skills, not a head, male, birthdate between 1916 and 1935, and any skill code of 3340. The appropriate PROFILE statement is:

```
SUBSET TITLE IS 'DEMAND REPORT';
GET PERSONNEL
IF [NO. SKILLS GT 2]
  AND    [LEVEL NE 'HEAD']
  AND    [SEX = 'M']
  AND    [EXTRACT (BIRTHDATE, 7, 2)
          GT 16 AND LT 35]
  AND    [ANY [SKILL (CODE) = 334ϕ]]:
  NAME, NUMBER, UNIT, CODE, SKILL
(NAME);
```

```
PROCEDURE WRITE.HEADING
  (ORG.UNIT, REPORTS.TO, ORG.NAME);
  BEGIN;
    FIELD = 'ORG UNIT';
    SPACE OUTPUT 5;
    FIELD = ORG.UNIT;
    SPACE OUTPUT BLOCK;
    FIELD = 'REPORTS TO';
    SPACE OUTPUT 3;
    FIELD = REPORTS.TO;
    SPACE OUTPUT BLOCK;
    SPACE OUTPUT BLOCK;
    FIELD = 'ORG NAME';
    SPACE OUTPUT 5;
    FIELD = ORG.NAME;
    SPACE OUTPUT BLOCK;
    SPACE OUTPUT BLOCK;
    FIELD = 'AUTHORIZED COMPLEMENT';
    SPACE OUTPUT BLOCK;
    SPACE OUTPUT BLOCK;
    RETURN;
  END;
```

```
''THIS PROCEDURE, WRITE.HEADING, HAS 3
''ARGUMENTS.
''IT FORMATS THE FIRST FOUR LINES OF THE
''PERIODIC REPORT, SUBSTITUTING ITS
''ARGUMENTS WHERE APPROPRIATE.
''THE OUTPUT DEVICE, AND THE DEFINITION
''OF A BLOCK HAVE BEEN DEFINED BY THE
''PROCEDURE WHICH CALLS IT.
''
''THIS LINE IS TYPICAL OF A STATEMENT
''WHICH INSERTS THE VALUE OF AN EXPRESSION
''IN THE OUTPUT STREAM.
''
''THIS LINE INSERTS 5 SPACES IN THE OUTPUT
''                                  STREAM.
''
''THIS LINE INSERTS AN ENTIRE LINE OF
''                                  SPACES.
''
''
''THIS STATEMENT RETURNS TO THE CALLING
''                                  PROCEDURE.
```

```
PROCEDURE WRITE.LINE                    ''THIS PROCEDURE, WRITE.LINE, IS SIMILAR
  (KEY,CODE,TITLE,QUAN,SAL);            ''TO THE PROCEDURE WRITE.HEADING.
  BEGIN;                                ''
    SPACE OUTPUT 5 + KEY * 5;           ''IT HAS FOUR ARGUMENTS, AND FORMATS MOST
    FIELD 4 = CODE;                     ''OF THE DATA LINES IN A PERIODIC REPORT.
    SPACE OUTPUT 1Ø - KEY * 5;          ''
    FIELD 32 = TITLE;                   ''THREE OF THE ARGUMENTS ARE DATA TO BE
    SPACE OUTPUT 2;                     ''INSERTED IN THE OUTPUT STREAM, BUT KEY
    FIELD 2 = QUAN;                     ''IS AN ARGUMENT USED TO CONTROL SPACING.
    SPACE OUTPUT 8;
    FIELD 6 = SAL;
    SPACE OUTPUT BLOCK;
    RETURN;
  END;

PROCEDURE WRITE.TOTALS                  ''THIS PROCEDURE, WRITE.TOTALS, IS SIMILAR
  (Q.TOTAL, S.TOTAL);                   ''TO THE PROCEDURE WRITE.HEADING.
  BEGIN;                                ''
    SPACE OUTPUT BLOCK;                 ''IT HAS TWO ARGUMENTS, AND FORMATS THE
    SPACE OUTPUT 19;                    ''LINE CONTAINING TOTALS
    FIELD = 'TOTAL';                    ''
    SPACE OUTPUT 29;
    FIELD 2 = Q.TOTAL;
    SPACE OUTPUT 18;
    FIELD 6 = S.TOTAL;
    SPACE OUTPUT BLOCK;
    SPACE OUTPUT BLOCK;
    RETURN;
  END;
PROCEDURE WRITE.AC ();                  ''THIS PROCEDURE, WRITE.AC, HAS NO
  BEGIN;                                ''ARGUMENTS. IT WRITES THE TITLE
    FIELD = 'ACTUAL COMPLEMENT';        ''ACTUAL COMPLEMENT
    SPACE OUTPUT BLOCK;
    SPACE OUTPUT BLOCK;
    RETURN;
  END;

PROCEDURE WRITE.DEVIATION               ''THIS PROCEDURE FORMATS THE DEVIATION
  (DEVIATION);                          ''FROM BUDGET LINE.
  BEGIN;                                ''IT HAS ONE ARGUMENT, THE DEVIATION.
    SPACE OUTPUT 19;
    FIELD = 'DEVIATION FROM BUDGET';
    SPACE OUTPUT 33;
    FIELD 6 = DEVIATION;
    SPACE OUTPUT PAGE;
    RETURN;
  END;

PROCEDURE WRITE.TITLE                   ''THIS PROCEDURE, WRITE.TITLE, WRITES A
  (REPORT.NAME);                        ''SIMPLE TITLE LINE.
  BEGIN;                                ''IT HAS ARGUMENT, THE NAME OF THE REPORT.
    SPACE OUTPUT 5Ø;
    FIELD = REPORT.NAME;
    FIELD = 'REPORT';
    SPACE OUTPUT BLOCK;
  END;

PROCEDURE PERIODIC.REPORT               ''THIS PROCEDURE, PERIODIC.REPORT, HAS TWO
  (LEVEL, N);                           ''ARGUMENTS: LEVEL, INDICATING WHETHER
```

```
 BEGIN;                                  ''REPORT IS TO COVER DIVISIONS,DEPARTMENTS
   VARIABLE T1, T2, T.TITLE, T.QUANT''', GROUPS, OR SECTIONS; AND N, INDICATING
          , T.SAL, T.CODE, T.UNIT,''HOW MANY UNITS TO REPORT ON.
          IND, TN, T3, INC, INC1, ''
          LIMIT, LIMIT1, U.TITLE; ''DECLARE VARIABLES.
   IND = Ø;                               ''
   TN = Ø;                                ''
   IF LEVEL = 'SECTION'; INC = Ø;         ''
   IF LEVEL = 'GROUP';                    ''INITIALIZE.
     BEGIN; INC = 9; INC1 = 1; END;       ''
   IF LEVEL = 'DEPARTMENT';               ''INCREMENTS HAVE TO DO WITH THE
     BEGIN; INC = 99; INC1 = 1Ø;END;''ASSIGNMENT OF NUMBERS TO UNITS.
   IF LEVEL = 'DIVISION';                 ''(PERSONNEL FILE HAS BEEN SORTED BY
     BEGIN; INC = 999; INC1=1ØØ;END;''                    UNIT NUMBER)
   OUTPUT TO PRINTER;                     ''DECLARE PRINTER AS OUTPUT DEVICE.
   DEFINE PAGE FOR OUTPUT AS 33*132;''DEFINE SIZE OF PRINTER PAGE.
   DO WRITE.TITLE(LEVEL);                 ''WRITE TITLE
   READ PERSONNEL; ELSE RETURN;           ''
ORG:READ ORGANIZATION; ELSE RETURN;       ''FIND ORGANIZATION UNIT OF APPROPRIATE
   IF ORG.LEVEL NE LEVEL; GO TO ORG;''LEVEL IN ORGANIZATION FILE.
   TN = TN+1;                             ''
   IF TN > N; RETURN;                     ''RETURN IF ENOUGH UNITS PROCESSED.
   LIMIT = ORG.UNIT+INC;                  ''
   DO WRITE.HEADING(ORG.UNIT,             ''WRITE HEADINGS. AUTHORIZED COMPLEMENT
     REPORTS.TO, ORG.NAME);               ''IS PRINTED FIRST.
   T1 = Ø;                                ''
   T2 = Ø;                                ''INITIALIZE TOTALS.
   T3 = Ø;                                ''
O.JOB:READ JOB; ELSE GO TO O.UNIT;        ''
   DO WRITE.LINE (Ø, JOB(CODE),           ''WRITE LINE OF REPORT FOR EACH JOB.
     JOB(TITLE), JOB(QUANT),              ''
     JOB(SALARY));                        ''
   T1 = T1 + JOB(SALARY);                 ''INCREMENT TOTALS.
   T3 = T3 + JOB(QUANT);                  ''
   GO TO O.JOB;                           ''
O.UNIT:READ UNIT; ELSE GO TO O.END;       ''WRITE LINE OF REPORT FOR EACH
   DO WRITE.LINE (1, UNIT(CODE),          ''SUBSIDIARY UNIT.
     UNIT(TITLE), 1, UNIT(SALARY));       ''
   U.TITLE = UNIT(TITLE);                 ''
   T1 = T1 + UNIT(SALARY);                ''
   GO TO O.UNIT;                          ''
O.END:IF LEVEL NE 'SECTION';T3='  ';      ''
   DO WRITE.TOTALS(T3,T1);                ''WRITE TOTALS FOR AUTHORIZED COMPLEMENT.
   DO WRITE.AC();                         ''
   T3 = Ø;                                ''BEGIN ACTUAL COMPLEMENT SECTION OF
MATCH:IF PERSONNEL(UNIT)=ORG.UNIT;        ''REPORT.
     GO TO P.JOB;                         ''FIND FIRST EMPLOYEE WHICH
   READ PERSONNEL; ELSE GO TO P.END;''BELONGS TO THIS UNIT.
   GO TO MATCH;                           ''
P.JOB:T.TITLE = PERSONNEL(TITLE);         ''
   T.CODE = JOB.CODE;                     ''
   T.QUANT = Ø;                           ''
   T.SAL = Ø;                             ''
P.JOB1:T.QUANT = T.QUANT+1;               ''
   T.SAL = T.SAL+PERSONNEL(SALARY);       ''
   READ PERSONNEL; ELSE GO P.JOB2;        ''GATHER INFORMATION ON ALL EMPLOYEES
   IF [PERSONNEL(UNIT)=ORG.UNIT]          ''IN UNIT WITH SIMILAR JOBS.
     AND [PERSONNEL(TITLE)=T.TITLE];''
     GO TO P.JOB1;                        ''
P.JOB2:DO WRITE.LINE(Ø, T.CODE,           ''
     T.TITLE, T.QUANT, T.SAL);            ''WRITE LINE FOR THIS JOB.
   T2 = T2 + T.SAL;                       ''
   T3 = T3 + T.QUANT;                     ''UPDATE TOTALS.
   IF PERSONNEL(UNIT)=ORG.UNIT;           ''
     GO TO P.JOB;                         ''DO NEXT JOB
P.UNIT:IF PERSONNEL(UNIT)>LIMIT;          ''
     GO TO P.END;                         ''
```

```
    T.SAL = ∅;                              ''
    T.UNIT = PERSONNEL(UNIT);               ''SUMMARIZE DATA FOR SUBSIDIARY UNITS.
    LIMIT1 = T.UNIT + INC1;                 ''
P.UNIT1:T.SAL = T.SAL+PERSONNEL(SALARY);
    READ PERSONNEL; ELSE IND = 1;           ''
    IF PERSONNEL(UNIT)<LIMIT1;              ''
      GO TO P.UNIT1;                        ''
    DO WRITE.LINE(1,T.UNIT,U.TITLE,1,       ''WRITE LINE OF REPORT FOR SUBSIDIARY UNIT
      T.SAL);                               ''
    T2 = T2 + T.SAL;                        ''
    IF IND = ∅; GO TO P.UNIT;               ''
P.END:IF LEVEL NE 'SECTION';T3='  ';        ''
    DO WRITE.TOTALS (T3,T2);                ''WRITE TOTALS AND DEVIATION.
    DO WRITE.DEVIATION (T2-T1);             ''
    GO TO ORG;                              ''PROCESS NEXT UNIT.
  END;                                      ''
Δ
```

Figure 5 – A Detailed PROFILE Example

The first line above titles the report; the second line indicates a simple retrieval operation; the IF phrase expresses a criteria for selecting objects to be retrieved; and the last line lists names of property values desired.

Sample output is shown in Figure 7.

APPENDIX II

Definitions

Blocks

A block in C-10 is a data structure providing bulk storage. Though individually small, blocks may be linked together to form sequences of data virtually infinite in length. All blocks reside permanently on disk. A set of block management operations allocates disk space to blocks, and locates temporary copies of blocks in core to permit changes or references to their contents.

Private stacks

Private stacks are a data structure provided in the C-10 system to store vectors of pairs of data. Each pair of data consists of a data element of arbitrary type, and a string (which may be null). Private stacks appear to be virtually infinite in length; they reside permanently on disk and are used only through a set of operative procedures which handle all storage allocation implicitly (via the block manipulators). Private stacks may be used as first-in-first-out stacks, last-in-first-out stacks, or association-lists as are found in some LISP implementations.

Streams

A stream is a compact sequence of data, virtually infinite in length and consisting of two basic components: data elements and markers. Markers are a well-ordered set of single characters which serve as both data separators and as data locators. As with private stacks, streams reside permanently on disk and are manipulated only through a set of operative procedures which hide all the problems of storage allocation from the users (via the block manipulators). Streams are commonly used to store textual information, but can also be used to store highly structured data.

Files are discussed in the text of the main section.

AUTOCODER – is the assembly language of the IBM 1410. An example of an AUTOCODER procedure, which computes factorials, written for the C-10 framework, is shown in Figure 8.

STEP – is a language similar to LISP. STEP operates on the C-10 data structures rather than on lists. A STEP procedure consists of a sequence of nested appeals to procedures in prefix form. The STEP procedure shown in Figure 9 happens to be one of the many executive routines that exist in C-10 (each user may write his own).

PROFILE – is discussed in the text of the main section.

```
                                                        GROUP REPORT

ORG UNIT      2110
REPORTS TO    2100

ORG NAME      SYST. ENGR. GROUP

AUTHORIZED COMPLEMENT

     1110             MANAGER                          1        17500
     5210             SECRETARY                        1         4200
          2111        SECTION                          1        52600
          2113        SECTION                          1        49CCO
          2115        SECTION                          1        42CCO

                      TOTAL                                                165300

ACTUAL COMPLEMENT

     1110             MANAGER, SYST. ENGR. GROUP       1        17500
     5210             SECRETARY                        1         4200
          2111        SECTION                          1        52300
          2113        SECTION                          1        48CCO
          2115        SECTION                          1        52500

                      TOTAL                                                174500

                      DEVIATION FROM BUDGET                                 9200

ORG UNIT      2120
REPORTS TO    2100

ORG NAME      STCS. ENGR. GROUP

AUTHORIZED COMPLEMENT

     1120             MANAGER                          1        14500
     5210             SECRETARY                        1         4CCO
          2122        SECTION                          1        39200
          2123        SECTION                          1        44500

                      TOTAL                                                102200

ACTUAL COMPLEMENT

     1120             MANAGER, STCS. ENGR. GROUP       1        14500
     5210             SECRETARY                        1         4CCO
          2122        SECTION                          1        30700
          2123        SECTION                          1        44500

                      TOTAL                                                 93700

                      DEVIATION FROM BUDGET                                -8500
```

Figure 6—Output Generated by PROFILE Example in Figure 5

```
                                                        DEMAND REPORT NO. 1
 1 = NAME    2 = NUMBER    3 = UNIT    4 = CODE    5 = NAME

1                     2       3      4      5
-------------------------------------------------------------------------

BOYD, W. V.           15052   2135
```

```
                                    1351  MECH TECHN
                                    3340  MACHINE OPR
                                    7320  MAINT TECH (E)
                                    3370  PAINTING OPR
COATES, C. L.          81130  2313
                                    3340  MACHINE OPR
                                    3345  MILLING MACH OPR
                                    3360  HEAT TREAT OPR
FLETCHER, M. W.        21475  2133
                                    1365  TOOL DESIGNER
                                    3125  TOOL MAKER
                                    3340  MACHINE OPR
                                    7320  MAINT TECH (E)
GORTON, R. A.          17590  2313
                                    3340  MACHINE OPR
                                    3345  MILLING MACH OPR
                                    7320  MAINT TECH (E)
GREEN, S. D.           34840  2313
                                    3340  MACHINE OPR
                                    3360  HEAT TREAT OPR
                                    7320  MAINT TECH (E)
LEVITT, P. S.          32924  2311
                                    3340  MACHINE OPR
                                    3345  MILLING MACH OPR
                                    3360  HEAT TREAT OPR
LITTLE, E. F.          42055  2311
                                    3340  MACHINE OPR
                                    3360  HEAT TREAT OPR
                                    3550  EXPEDITER
MILLER, F. L.          52467  2311
                                    3340  MACHINE OPR
                                    3510  ASSEMBLER
                                    7320  MAINT TECH (E)
                                    3570  DISPATCH ASST
```

Figure 7—Output Generated by Simple PROFILE Example in Text

```
            TITLEFACTORIAL
            HEADRFACTORIAL PROCEDURE
            BEGINQFACTORIAL,001,001        DECLARE 1 ARGUMENT, 1 TEMP
ARG         EQU  QT1                       ASSIGN ARG SYMBOLIC NAME
TEMP        EQU  QT2                       ASSIGN TEMP SYMBOLIC NAME
            COMP ARG,ZERO                  COMPARE ARGUMENT, 0
            BU   RETURNONE                 BRANCH UNEQUAL
            MLC  ARG,TEMP                  SAVE ARG
            SUB  Q1,ARG                    CALL SELF RECURSIVELY
            CALL QFACTORIAL,,ARG           WITH ARG-1, RESULT IN QVALUE
            CALL QMUL,,TEMP,QVALUE         MULTIPLY, RESULT IN QVALUE
            RETRNQVALUE                    RETURN RESULT
RETURNONE   EQU  *&X1                      NOTE RELOCATION BY INDEX 1
            RETRNONE                       RETURN 1
            DCW  '000000C0C0I'
ZERO        EQU  *&X1-1                    DECLARE CONSTANTS
            DCW  'C000000001I'
ONE         EQU  *&X1-1                    NOTE RELOCATION BY INDEX 1
            FIN
            END
```

Figure 8—A Sample AUTOCODER Procedure

```
(DEFINE ((GSEXEC (FUNCTION ()                    ''DECLARE GSEXEC FUNCTION, NO ARGS
  (PROG (EDITOROUTPUT)                           ''DECLARE VARIABLE EDITOROUTPUT
    LABEL (PRINT (TIME))                         ''PRINT TIME USING TIME FUNCTION
          (PRINT (DATESG1))                      ''PRINT DATE USING DATESG1 FUNCTION
    (ASSIGN EDITOROUTPUT (EDITOR                 ''USE EDITOR TO GET AND EDIT MESSAGE
                          NIL NIL 'OK'))         ''EDITED MESSAGE IN EDITOROUTPUT
    (IF (EQUAL (SUBGRP EDITOROUTPUT)             ''CHECK FIRST ATOM OF MESSAGE FOR
               'ENDGS')                          '''ENDGS'
        (RETURN))                                ''RETURN TO CALLING PROCEDURE IF SO
    (IF (EQUAL (SUBGRP EDITOROUTPUT)             ''CHECK FIRST ATOM OF MESSAGE FOR
               '(' )                             ''LEFT PARENTHESIS
        (PROG ()                                 ''IF LEFT PARENTHESIS, PROCESS
          (PRINT (STEPP EDITOROUTPUT))           ''MESSAGE WITH THE STEPP PROCESSOR
          (GO LABEL)))                           ''AND GO THROUGH CYCLE AGAIN
    (PRINT (STEPP (TRANP EDITOROUTPUT)))         ''ELSE MESSAGE IS PROFILE - PROCESS
    (GO LABEL)))))                               ''USING TRANP AND THEN CYCLE AGAIN
```

Figure 9—A Sample STEP Procedure

Handling the growth by definition of mechanical languages*

by SAUL GORN
The Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

It has for many years been my opinion that programming languages in particular, and mechanical languages in general, exhibit many phenomena generally thought to be characteristic of natural languages.

The concept of mechanical language, in my mind, includes all sorts of notational and signalling systems, whether one-dimensional or multi-dimensional, sequential or simultaneous acting, continuous or discrete, descriptive or prescriptive; it is also supposed to subsume a vast variety of recording media subjected to a vast variety of sensing devices (including biological as well as physical), with a vast variety of permanence characteristics, reaction times, storage arrangements in space-time, retrieval devices, coupling, coding, and translational processors.

Thus musical and choreographic notations are mechanical languages, as are structural formulae notations in chemistry, nucleotide chain diagram schematics in genetics, parsing tree and other phrase marking systems in linguistics, and mechanical drawing, wiring diagram, block diagram, production control charts and organization chart systems in industry. So are cataloguing, inventory and accounting systems, whether used in industry, commerce, war, government, or religion. I would even include notations and arrangement systems for monuments and their contents.

*The Research behind this paper was made possible by the joint support of the Army Research Office (DA-31-124-ARO(D)-98), the National Science Foundation (NSF-GP-5661), and the Public Health Service Research Grant (5 RO1 GM-13,494-02) from the National Institute of General Medical Sciences. It was presented in a preliminary report at the Systems and Computer Science Conference, University of Western Ontario, London, Ontario, September 10 and 11, 1965, and the proofs of the theorems and the presentation of the processors discussed here appear in the Proceedings of that conference, under the title "Explicit Definitions and Linguistic Dominoes."

The time scales and space scales therefore vary from milli-microseconds to millenia, from microns to light years; the adaptation of devices (from artifacts to specially educated humans) working in different time scales, different transfer rates, different reaction times, with different signalling languages, and the logic of their system assembly, even apart from adaptability in time, is part of the problem of the study of mechanical languages.

The more successful a mechanical language (system) is in its ease of specification for many different types of areas, and in its adaptability among different kinds and levels of artifacts and humans (i.e. processors), the denser will be its usage for communication among men, among machines, and between men and machines. It will therefore tend to 'grow,' to reach for 'universality' as a 'common language.'

An example of such a trend appears if one looks at the history of mathematical notation. As long as very few people operated with numbers (as abstracted from things) the notations for numbers were primitive, but stable; so stable that they lasted for millenia. Only a specialized historian would have been able to perceive their process of change. The urban revolution increased their usage, and therefore their rate of change; their stability was reduced from millenia, to centuries, to decades. The arithmetic notation expanded into the arabic number representation language, but the users could still deceive themselves into crediting the changes to ingenious individuals rather then recognizing them as part of a social phenomenon. This continued even into the development of algebraic notation and its growth into a notation for analytic expression. But when the notation grew to include symbolic logic and operations with sets, and the applications spread through all the sciences and professions, the contributions to the structure of the mechan-

ical language became too numerous to be anything but anonymous, and therefore social. Furthermore, the logical operations became devices by which the language itself could be examined; the human activity became self-conscious, and the language became self-referencing and capable of operating, in the same fashion, many meta-syntactic levels (I have called such, 'languages with unstratified control'). The tendency is now for this mechanical language with its descriptive precision and its ability to analyze so much one-dimensional symbol manipulation to fuse with programming languages with their prescriptive precision and their ability to synthesize so much of the processing of that same area of one-dimensional, and even multi-dimensional symbol manipulation.

Thus, different mechanical languages may therefore undergo *fusion* as time goes on.

But a particularly successful language will also tend to reflect the behavior of the human community by undergoing *fission* by splitting into dialects, even separate languages. This will surely happen if the using community splits into groups which seldom intercommunicate, and each of which restricts itself to a favorite subsystem of expressions; this will first be a jargon, then a dialect, then a separate language. In programming, these jargons are favorite systems of macro-instructions, introduced by the very macro-definition facility which made the base language so flexible and 'universal'.

In any event, we are now more and more concerned with growing languages, extended machines, growing machines, self-expandable languages, user-expandable languages and the like. (See Cheatham,[3] Carr,[2] Otrand[15] and Galler & Perlis.[7])

How are we to control the growth?

If it is completely uncontrolled and is allowed to 'just grow', such a man-machine, multi-lingual and multi-levelled processor and language system can die either by choking on the sheer bulk of what it swallows without rejecting or reforming, or it can die by committing mitosis, giving birth to a number of independent such subsystems with practically no intercommunication among them.

However, a natural control of the growth of such a system will have to be the result of continuing and never-ending work by hundreds of people. Each one will have to be able to refocus his attention from the grandiose to the picayune when necessary, without losing sight of the big picture. It is impossible to say whether it is the big wheel or the tiny wheel in such a system which is really connected to the ground. The same remark applies to the people in it.

Therefore, this paper will attempt only to suggest how such growth may be understood and controlled, while restricting itself to a very special category of mechanical languages with a very strong structure, and which grows by a very limited means called 'explicit definition'.

Categories and forms

Let me illustrate the category of languages to which we are limiting our discussion by tracing one simple algebraic expression through a number of its languages. We do this not merely to show how general our considerations are, by illustrating how big the category might be; we do it because the category is a fusion of many languages, rather than only a set, and those concepts and devices are to receive special attention which have a correspondent in each language of the category.

Consider, then, how the simple algebraic expression 'ab + b(c+d)', belonging to a one-dimensional language with which most of us are very familiar, transforms into seven other expressions. (See fig. 1).

It is the comparison among corresponding expressions in different languages of the category which prompts us to make the following decisions:

1. there is an alphabet of 'objects' which includes symbols like a,b,c,d,+, and something called 'times', even though examples 1 and 3 have no separate symbol for it, and even though examples 2, 4 and 8 represent it by ×, where 5, 6 and 7 use *.
2. there is an alphabet of 'context signals', much more concerned with 'how' the expressions are to be read than with 'what' they contain, and consequently more variable among the languages of the category. Examples are parentheses, punctuation marks, connecting lines, and tabulation lines. These are 'control characters'.
3. there is an 'addressing system'; only the object characters have addresses; control characters only help us to find these 'addresses', but are not themselves assigned addresses. The corresponding objects in the corresponding expressions have the 'same' addresses.

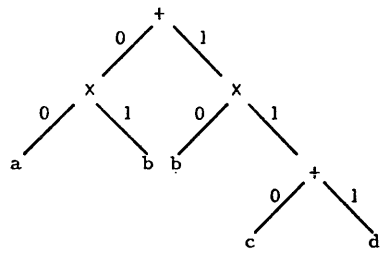
We therefore say that we really have a category of 'language functions', each applicable to many '(object) alphabets' to form the languages in the category. The control characters are really the marks belonging to the language functions, and often are replaced by, or replace, the scanning process.

Thus we might call the 'common' alphabet used here $\mathscr{A}$ = {a,b,c,d,+....}, the language functions $\mathscr{F}\mathrm{T}_{\mathrm{pd}}$, $\mathscr{F}$, $\mathscr{P}$, and $\mathscr{M}$ to correspond to expressions 2,4,5,6,7, and 8. Thus the tree in 2 belongs to the language $\mathscr{T}_{\mathscr{A}}$. If symbols from logic form an object alphabet $\mathscr{B}$ = {∨, ∧, ~, ⊃, ≡, p, q, r....}, then ⊃p⊃qp might be an expression from the language $\mathscr{P}_{\mathscr{B}}$

1.- ab + b(c + d)

left-to-right;precedence to multiplication

2.-



Tree form

3.- (d + cb) + (ba)

right-to-left; precedence to +

4.-

address	character
*	+
0	x
1	x
00	a
01	b
10	b
11	+
110	c
111	d

Function tabulation

5.- + * * a b b + c d (Depth)

6.- +(*(a, b), *(b, +(c, d))) (Functional)

7.- +*ab*b+cd (Prefix)

8.- *, +; 0, x; 1, x; 00, a; 01, b; 10, b; 11, +; 110, c; 111, d. (Address mapping)

Figure 1 – Corresponding expressions in eight languages of a category

The two-dimensional language functions seem to have some advantages in human communication (e.g. 𝒹 and T in 2 and 4); those, like 4 and 8, which are explicit about the mapping from addresses to characters, seem to have some advantages when we seek to establish a mathematical theory; those like $\mathscr{P}_d$, $\mathscr{P}$, and $\mathfrak{p}$ in 5, 6, and 7 seem to have some advantages when it comes to designing efficient processors. For example $\mathfrak{p}$ has an especially efficient 'scope analyzer' (see Gorn[11]), where $\mathscr{P}_d$ is more efficient for 'depth analysis'.

The first restriction we make is therefore to such categories that corresponding expressions in the different languages all have the same 'parsing tree', and are therefore uniquely determined by systems of 'addresses' called 'tree-domains' and the corresponding objects to be found there. Thus, whatever else we may find in the category, we can always depend on the representations in languages like those illustrated in 2, 4 and 7 of fig. 1.

The second restriction that we make is that the languages in the category be 'saturated' in the sense that 'any validly parsed expression' actually belongs to the system. We express this by demanding, first, that the system of language functions be applicable to any and all 'simply-stratified alphabets'. This means that each character have a kind of 'order', called its 'stratification number', much like the number of independent variables possessed by a numerical function. The second demand made by the restriction is that any expression obtained from a tree by putting characters of stratification n at nodes of ramification n (i.e. of 'exiting' order n) is a valid expression in the system.

This second restriction means that the prefix language form (type 7) is to be 'complete'. It is this language function which we have called the 'complete prefix language function'. (In Gorn[12] we show that it is also saturated in that the addition of one more word to its extent would cause it to cease being 'uniquely deconcatenable'. See also[9].) Naturally all the other languages of the category have the corresponding kind of completeness.

The only reason we make such a point of this second restriction (It is unduly harsh; in ordinary algebraic notation, for example, we do not demand that the expression '(a=b) + (c=d)' be meaningful.) is that we need it as a guide in our third restriction. We will allow our category to 'grow' by 'explicitly defining'

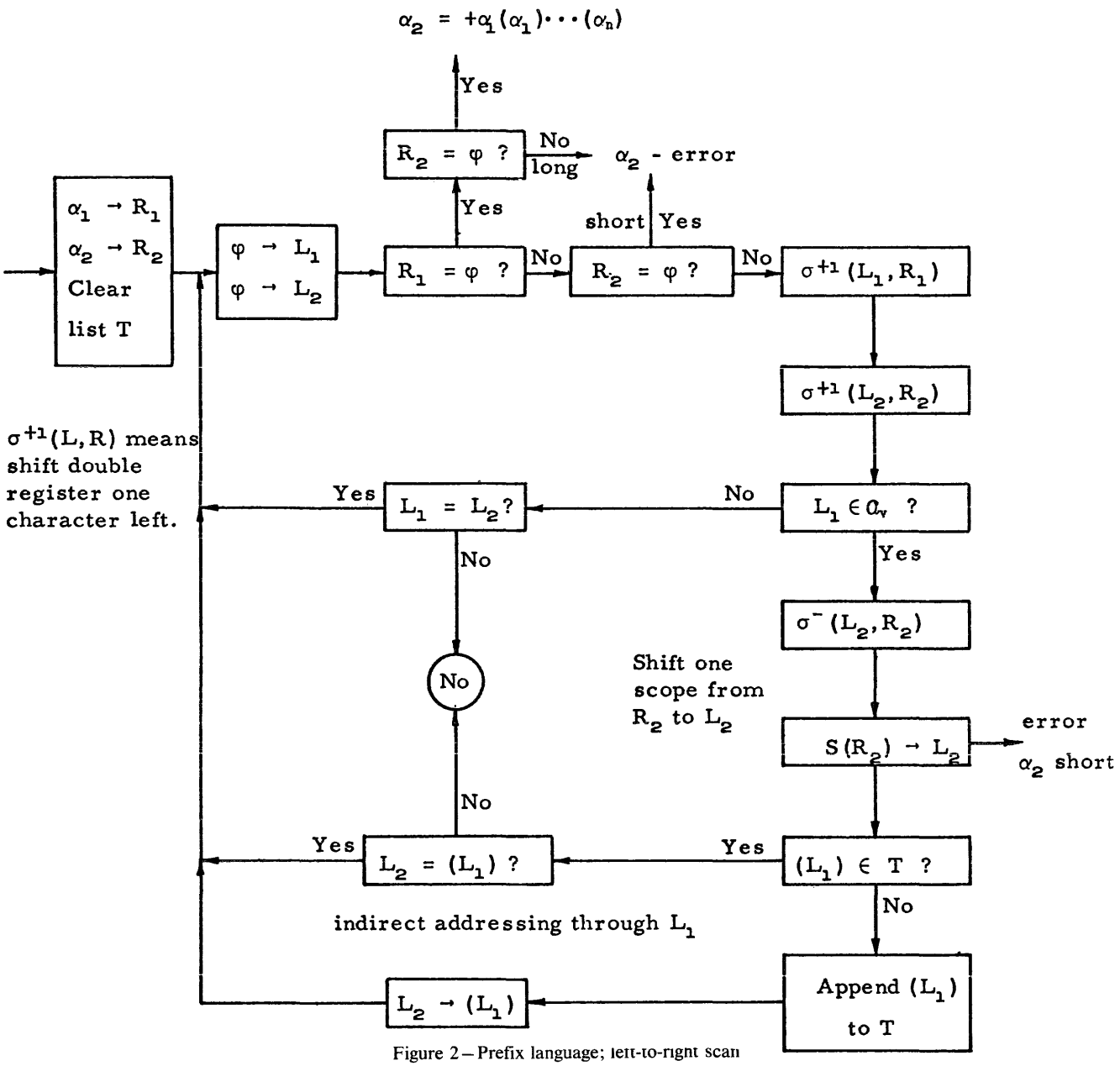


Figure 2 – Prefix language; left-to-right scan

new characters in the object alphabet. Our third restriction limits the form of these explicit definitions by demanding that the extended language category be also complete. Let us see what this requires.

Suppose 'd' is the newly defined object character. It will have to possess, by our first restriction, a fixed stratification number, say n. If, then, α_1, α_2, ..., α_n designate any good expressions of $\mathscr{P}\mathscr{A}$ by our second restriction $\mathscr{P}\mathscr{A}$ must also contain a good expression of the form '$d\alpha_1\alpha_2...\alpha_n$'. Thus, the explicit definition must have the 'form' $\beta_1 \underset{df}{=} \beta_2$, where the 'definiendum' β_1 must have the 'simple form' $d\alpha_1\alpha_2...\alpha_n$, and the 'definiens' β_2 must not contain the character 'd' explicitly (no recursion), and must be an obvious syntactic function of the syntactic variables $\alpha_1,...,\alpha_n$ for all languages of the category. By the 'principle of syntactic invariance' (see Gorn[10]), the α_i must be variables representing any good expressions throughout the category, because they are appropriate 'scopes' and the completeness of the category means that all good 'scopes' are 'good words' and vice versa.

We must therefore be able to define and recognize forms as well as expressions. This is easily done without too much work by introducing an alphabet of 'scope variables': $\mathscr{A}_v = \{\alpha_1\ \alpha_2,...\}$ having no characters in common with $\mathscr{A}$, and each character of which has stratification zero. If $\mathscr{A}' = \mathscr{A} \cup \mathscr{A}_v$, then $\mathscr{P}\mathscr{A}$ is a form language, representing a whole category of form languages in which 'tree forms' can be generated, recognized, and compared.

For example, we have the following definition throughout the category of languages of tree patterns:

Definition: The form β_1 is said to be 'of the form β_0', and β_0 is called 'an initial pattern of β_1' (we write $\beta_0 \leqslant \beta_1$), if there is a complete independent set of scopes $\sigma_1, \ldots, \sigma_n$ of β_1, where n is the number of end-points of β_0, such that, if $\alpha_i = \alpha_j$ in β_0, then $\sigma_i = \sigma_j$, and such that $\beta_1 = \beta_0\ (\sigma \rightarrow \alpha_1, \ldots, \sigma_n \rightarrow \alpha_n)$. We can also write: $\beta_1 = +\beta_0\sigma_1 \ldots \sigma_n$.

If we think of the variables in $\mathscr{A}_V$ as also referring to some unspecified addressing system for the control of storage of expressions, then this definition is effective; figure 2 presents the design of the processor called a 'tree pattern recognizer'.

A number of other notations might be mentioned as being convenient both in the theory and in the construction of processors:

a. 'Caβ_1' to mean 'the character at (tree) address a in tree (form) β_1'.
b. 'Saβ_1' to mean 'the scope at (tree) address a in tree (form) β_1'.
c. 'Dβ_1' to mean 'the domain of (tree) addresses in tree (form) β_1'.
d. If a_2 and a_3 are tree addresses, then $a_1 = a_2 \cdot a_3$ can be read as 'the tree address of relative address a_3 with respect to address a_2', and a_2 is a 'tree-predecessor' of a_1 : $a_2 \leqslant a_1$. Similarly, if A_1 and A_2 are two sets of tree-addresses, $A_1 \cdot A_2$ will be the set of tree-addresses of relative address some A_2 with respect to some A_1; it is to be interpreted as the null set Λ if either A_1 or A_2 is null.
e. We can also permit the scope-operator 'S' to refer to tree-domains as well as to trees. Thus, if D is a tree domain and $a_1 \in D$, then there is a tree domain D_1 such that $Sa_1 D = a_1 \cdot D_1$; also, $Sa_1 \cdot a_2\, D = a_1 \cdot Sa_2 D_1$. This notation sets up a primitive address computation facility (in fact a semi-group with unit '*' meaning root-address) for tracing and identifying the effects in expressions of growth by definition in the category.
f. 'an occurrence of the character $c \in A$ in β' will mean 'an address a such that $c = C\ a\ \beta'$, and the set of such occurrences can be written '$C^{-1}\ c\ \beta'$. Thus '$a \in C^{-1}\ c\ \beta'$ means 'a is an occurrence of c in β'.

It is now possible to recognize internal patterns as well as initial patterns in trees: β_0 is an internal pattern of β_2 occurring at the tree address a if $\beta_0 \leqslant S\ a\ \beta_2$; in this case we also write $a \in C^{-1}\beta_0\beta_2$, and speak of 'an occurrence of the form β_0 within β_2'. If β_0 is deeper than zero, i.e. is more than just one object character at the root, then we say that the form β_1 'dominates' the form β_2 with the intermediary β_0 whenever β_0 s a scope of β_1 and the initial pattern of β_2, but β_0, β_1 and β_2 are not all the same; if we replace that occurrence of β_0 in β_1 by β_2, the result $\beta_1(\beta_2 \overset{a}{\rightarrow} \beta_0)$ if $a \in C^{-1}\beta_0\beta_1$, is a '$\beta_1\beta_2$-domino'.

We now have the apparatus by which we can specify an 'explicit definition', and trace its effect by means of a primitive kind of computation with tree addresses.

EXPLICIT DEFINITIONS AND TREE MAPS

We have now fixed upon an appropriate definition of 'explicit definition' for our categories of mechanical languages. It is one in which:

a. The definiendum is a simple form, i.e. either of depth zero (hence only the single new character), or of depth one with only distinct variables at the end-points,

b. the definiens is an arbitrary form φ_d, at least as deep as the definiendum (hence $\neq \alpha$), not containing d explicitly, but each variable of which does occur again in the definiendum.

Ordinarily explicit definitions are introduced in order to say in less space something one will want to repeat fairly often. In other words, a very common motive for an explicit definition is the desire to have a large family of abbreviations; each definition provides an infinite set of abbreviations, all of the same form. (Thus each definition is like an 'optional linguistic transformation'.) Because the size of a tree can be measured both in depth and in breadth, two of the simplest types of explicit definition are the pure depth reducer, and the pure breadth reducer:

An explicit definition, $d\alpha_1 \ldots \alpha_n \underset{df}{=} \varphi_d$, is a 'pure depth reducer' if those end-points of φ_d which are variable are, reading from left to right, in $\mathfrak{p}$ or $\mathfrak{z}$ form, exactly $\alpha_1, \alpha_2, \ldots, \alpha_n$, and at least one variable has depth greater than one. If $n = 0$, so that $d \underset{df}{=} \varphi_d$ where no end-point of φ_d is variable, we do not call it a depth reducer, no matter how deep φ_d may be. An example of a pure depth reducer is $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_1 c c_0 \alpha_2$.

An explicit definition is a 'pure breadth reducer' if the depth of every variable in φ_d is one, and those end-points of φ_d which are variable are, reading from left to right, in $\mathfrak{p}$ or $\mathfrak{z}$ form and ignoring repetitions, exactly $\alpha_1, \alpha_1, \ldots, \alpha_n$, and at least one variable occurs more than once. The same interpretation is made if $n = 0$ as before; no matter how broad φ_d may be, it is not a breadth reducer. An example of a pure breadth reducer is $d\alpha_1\alpha_2 = c\alpha_1 c c_0 c_0 c_0 c_0 \alpha_2 \alpha_1$.

More generally, if φ_d is the definiens of an explicit definition, let $A_{di} = C^{-1}\alpha_i\varphi_d$ (the occurrences, possibly

$$d\,\alpha_1\,\alpha_2 \underset{df}{=} c\,\alpha_2\,c\,\alpha_1\,\alpha_2$$

Definiendum

address	character
*	d
0	α_1
1	α_2

Definiens

address	character
*	c
0	α_2
1	c
10	α_1
11	α_2

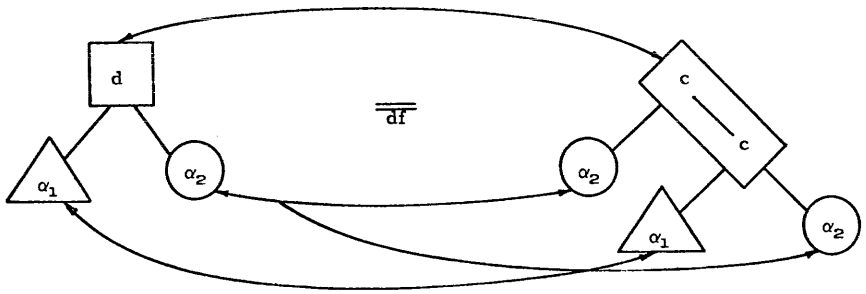


Figure 3 – An explicit definition

null, of α_i in φ_d), and $A_{do}=D\varphi_d - \bigcup_{i=1}^{n} A_{di}$ (the 'interior' of φ_d); let m_i be the number of addresses in A_{di}, which we could call the 'breadth' or 'multiplicity' of α_i, and let d_i be the maximum depth of α_i in A_{di}. Then the definition does some breadth reduction if $m_i>1$ for at least one i, and it does some depth reduction if $d_i > 1$ for at least one i.

If $m_i = 0$ for some $i \leq n$, the stratification of d, then the definition is allowing arbitrary scopes to be introduced, and we say that the definition is a 'scope-expander'. An example of a 'pure scope-expander', i.e. one in which neither depth or breadth reduction occurs, is $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2$, or $d\alpha_1\alpha_2 \underset{df}{=} cc_0$.

Finally, an explicit definition may be introduced neither to expand scopes, nor to reduce depths or breadths, but simply to permute scopes. In a pure permuter' $m_i = 1$ and $d_i = 1$ for $i = 1, \cdots, n$. Thus $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2 cc_0c_0c_0\alpha_1$ is a pure permuter.

Most definitions we might envision would be mixtures of these types, and for each such mixture we might imagine each type of effect to be introduced by a separate 'ideal definition', thereby 'factoring' the definition into a sequence of pure types. For example, the explicit definition $d\alpha_1\alpha_2 \underset{df}{=} c\alpha_2 c\alpha_1\alpha_2$ might be envisioned as the result of

1. a pure depth reducer $d_1\alpha_1\alpha_2\alpha_3 \underset{df}{=} c\alpha_1 c\alpha_2\alpha_3$,
2. a pure breadth reducer $d_2\alpha_1\alpha_2 \underset{df}{=} d_1\alpha_1\alpha_2\alpha_1$,
3. a pure permuter $d\alpha_1\alpha_2 \underset{df}{=} d_2\alpha_2\alpha_1$.

Figure 3 illustrates this definition.

We can now be precise about the meaning of such expressions as:

a. The elimination of an occurrence of d in a form β_2: if $a \in C^{-1}d\beta_2$, and the scopes $\sigma_i = Sa{\cdot}i\beta_2$, and ψ_1 is the form obtained from φ_d by replacing each α_i by σ_i (in other words, ψ_1 is the application of the definiens form to the scopes of the occurrence of d in β_2), i.e. $\psi_1 = \varphi_d(\sigma_1 \rightarrow \alpha_1, \ldots, \sigma_n \rightarrow \alpha_n)$, then the replacement of the scope at a of β_2 by ψ_1 yields an expression $\beta_1 = \beta_2(\psi_1 \rightarrow Sa)$ which is said to be obtained from β_2 by the elimination of d at a; we also write

$$\beta_2 \overset{E_d}{\underset{a}{\rightarrow}} \beta_1, \text{ and } \beta_1 = E_d\, a\beta_2,$$

and call β_1 an immediate d-descendent of β_2, thereby beginning the definition of a partial ordering relation among expressions: $\beta_2 \underset{d}{\geqslant} \beta_1$.

b. The introduction of d in a form β_1 at an occurrence of φ_d: if $a \in C^{-1}\varphi_d\beta_1$, so that $\varphi_d \leq Sa\beta_1$, i.e. φ_d

is an internal pattern in β_1, then there is a complete independent set of scopes $\sigma_1,\ldots, \sigma_n$, occurring with the repetitions corresponding to the α_i in φ_d, at addresses in β_1 whose relative addresses with respect to a equal the address of the end-points α_i in φ_d; let β_0 will say that β_2 is obtained from β_1 by introduction of

d at a; we also write $\beta_1 \xrightarrow[a]{I_d} \beta_2$ and $\beta_2 = I_d a \beta_1$; β_2 is an immediate d-predecessor of β_1 and $\beta_1 \underset{d}{\leqslant} \beta_2$.

Clearly $I_d a$ and $E_d a$ are inverse operators.

c. In general, $\beta_2 \underset{d}{\geqslant} \beta_1$ and $\beta_1 \underset{d}{\leqslant} \beta_2$, if either $\beta_2 = \beta_1$ or there is a chain of d-eliminations $\beta_2 = \beta_{0\,1}$

$$\longrightarrow \underset{a_1}{E_d}\ \underset{a_2}{E_d} \ldots \beta_{0n-1} \xrightarrow[a_{n-1}]{E_d} \beta_{0n} = \beta_1.$$

In this case we say that β_2 is a d-predecessor of β_1, β_1 is a d-descendent of β_2, and β_1 is obtained from β_2 by d-reduction. Figure 4 presents the design of a recognizer of $\beta_2 \underset{d}{\geqslant} \beta_1$.

d. β_2 is d-equivalent to β_1, and we write $\beta_2 \underset{d}{\equiv} \beta_1$ if the chain of operations, as in c, may contain any combination of d-introductions and d-eliminations.

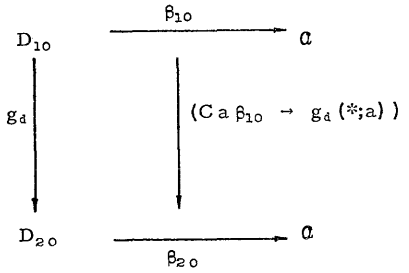
It is easy to see that $\underset{d}{\leqslant}$ is a partial-ordering, and $\underset{d}{\equiv}$ is an equivalence relation. We will designate the set of all d-descendents of β_0 by $L(\beta_0)$, and the d-equivalence class of β_0 by $E(\beta_0)$.

If $\beta_2 \underset{d}{\leqslant} \beta_1$, there might be many elimination chains leading from β_2 to β_1; in spite of this, for each character occurrence in β_1, there is only one occurrence in β_2 'responsible for it'. This is shown by exhibiting a uniquely determined mapping from β_1 *into* β_2.

Figure 3 has already shown this in the critical case where $\beta_{10} = \varphi_d$ and $\beta_{20} = d\alpha_1 \ldots \alpha_n$, here $CA_{do}\,\beta_{10} \rightarrow d = C^*\beta_{20}$, and $CA_{di}\beta_{10} \rightarrow \alpha_i = Ci{-}1\beta_2$. This is the 'generating d-map', $g_d(*;a)$, which performs a linguistic transformation from β_{10} into β_{20}, where each of these expressions can also be looked upon as mappings, called trees, from tree-domains into the object alphabet; these linguistic transformations are therefore representable as composite mappings fulfilling a commutative condition:

$$(Ca\beta_{10} \rightarrow g_d(*;a)) \circ \beta_{10} = \beta_{20} \circ g_d(*;a),$$

a relation we might choose to represent by the symbolism $\beta_{10}\ E\ \beta_{20}\ g_d$, or even by the diagram extending the type used by algebraic topologists:



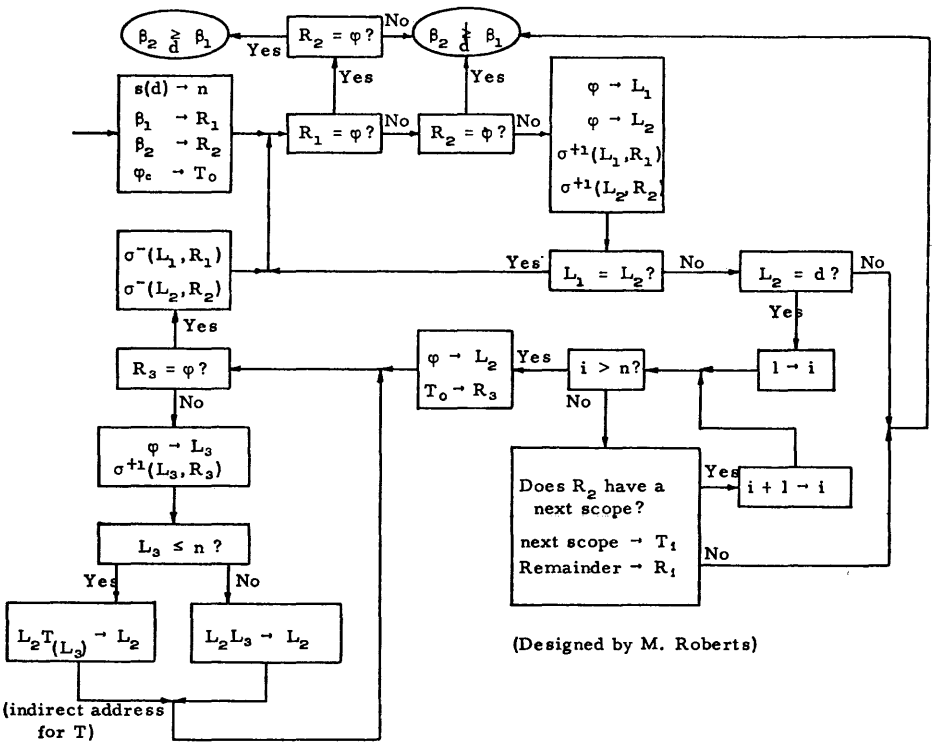


Figure 4 – Recognizer for the relation $\beta_2 \geqslant \beta_1$

We can extend such mappings from occurrences of d and φ_d in β_{20} and β_{10} alone by first defining the address map:

$$g_d(a_1;a) = \begin{cases} a \text{ if } a_1 \leq a \\ a_1 \text{ if } a \in a_1 \cdot A_{do} \\ a_1 \cdot i\text{-}1 \cdot a'' \text{ if } a \in a_1 \cdot A_{di} \cdot a'' \end{cases}$$

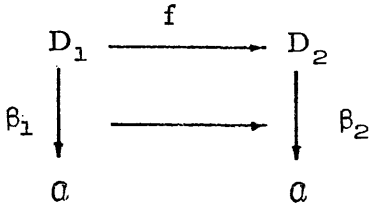
and then considering composites of such address maps, f, which we call d-maps.

The design of the recognizer of the relation $\beta_2 \underset{d}{\geq} \beta_1$, shown in figure 4, depends upon the:

Theorem: The following conditions are equivalent:

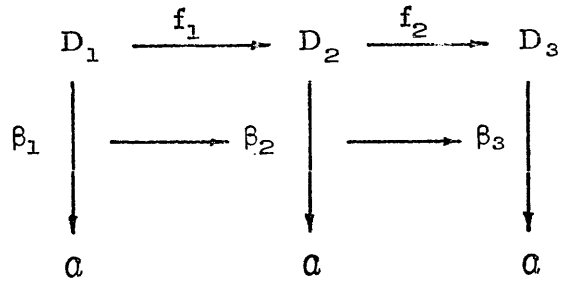
1. $\beta_1 \in L(\beta_2)$,
2. $\beta_2 \underset{d}{\geq} \beta_1$,
3. There is a d-map of β_1 into β_2,
4. There is a unique d-map of β_1 into β_2.

The generalized d-map can therefore be symbolized by $\beta_1 \overset{E}{\sim} \beta_2 \circ f$, or by the diagram:



and the composition of d-maps can be reflected in such notation as:

$\beta_1 \overset{E}{\sim} \beta_3 \circ (f_2 \circ f_1)$, or



If, furthermore, the explicit definition is not a scope expander, and $\beta_2 \underset{d}{\geq} \beta_1$, the uniquely determined mapping from β_1 into β_2 is a mapping 'onto'.

In any case, such mappings f are uniquely determinded by addresses only; i.e. by a computation which given any $a \in D_1 = D\beta_1$, yields $f(a) \in D_2 = D\beta_2$.

Actually, an even stronger statement can be made; these address maps are even independent of the particular trees. $E_d a$ is a functor applicable to any tree β_2 for which $C a \beta_2 = d$, and it is possible to compute $g_d(a;b)$ for any address without knowing anything else about β_1 beyond the fact that $a \in C^{-1}\varphi_d\beta_1$. We can therefore talk, within the context of a fixed explicit definition, of 'the elimination functor <a>'.

Furthermore, $E_d a_1 E_d a_2$ will be applicable, as a functor, to an infinite number of expressions, if to any at all; first of all, if it is applicable to β, then $C a_2 \beta = d$; secondly we must have $d = C a_1 E_d a_2 \beta$. If we let:

$$A_{di} = \{a_{i1}, a_{i2}, \ldots, a_{imi}\},$$

where m_i is the multiplicity of α_i, such an occurrence of d at address a_1 is a priori impossible only if $g_d(a_2;a_1) \in A_{do}$; in other words it is possible only if $a_1 \geq a_2 \Rightarrow (\exists i,j)(a_1 \geq a_2 \cdot a_{ij})$. Let the relation $a_1 \geq a_2$ among addresses mean just this, so that our condition reads: $a_1 \geq a_2 \Rightarrow a_1 \geq a_2$. This, then, is precisely the condition under which the 'elimination functor $<a_1,a_2>$' exists; it is applicable to any one of the infinite number of expressions β with an occurrence of d at a_2 and also at $g_d(a_2;a_1)$. We could similarly validate and identify the 'elimination functor $F = <a_1, \cdots, a_n> = E_d a_1 \cdots E_d a_n$' and restrict ourselves to a primitive address computation as in Figure 5.

$d\alpha_1\alpha_2 \overset{df}{=} c\alpha_2 c\alpha_1\alpha_2$

$$g_d(*;a) \begin{cases} <*> \{*,0\} = \{10\} \\ <*> \{*,1\} = \{0,11\} \end{cases}$$

$$\begin{aligned} <111,0,*>\{*,1,11\} &= <111,0>\{<*>\{*,1\} \cup <*>\{*,1\cdot 1\}\} \\ &= <111,0>\{\{0,11\} \cup \{0,11\}\cdot 1\} \\ &= <111,0>\{0,11,01,111\} \\ &= <111>\{<0>\{0,01\} \cup <0>\{0,11,111\}\} \\ &= <111>\{0\cdot <*>\{*,1\} \cup \{11,111\}\} \\ &= <111>\{0\cdot\{0,11\} \cup \{11,111\}\} \\ &= <111>\{00,011,11,111\} \\ &= <111>\{111,00,011,11\} \\ &= \{00,011,11\} \end{aligned}$$

Figure 5 – An address computation for d-elimination

The problem solved by the computation in figure 5 could be stated as follows: If we explicitly define d by means of $d\alpha_1\alpha_2 = c\alpha_2 c\alpha_1\alpha_2$, then we would like to know what 'traces' are left from original occurrences at {*,1,11} in any word to which the following elimination functor is applicable; d is eliminated at *, then at 0, and then at 111. According to the computation, the only traces will be at addresses 00, 011, and 11. In particular, if d's occurred originally only at *, 1, and 11, they would not be completely eliminated; there would still be occurrences of d at 00, 011, and 11.

A study of Figure 5 reveals that the address computation is driven to a conclusion by an algorithm which applies the following rules:

Rule 1: $<*> \{*\} =$

Rule 2: If $i < s(d)$ (the stratification of d), then for every address $a <> \{*,1 \cdot a\} = A_{di} \cdot a = \{a_{i1} \cdot a, \ldots, a_{imi} a\}$.

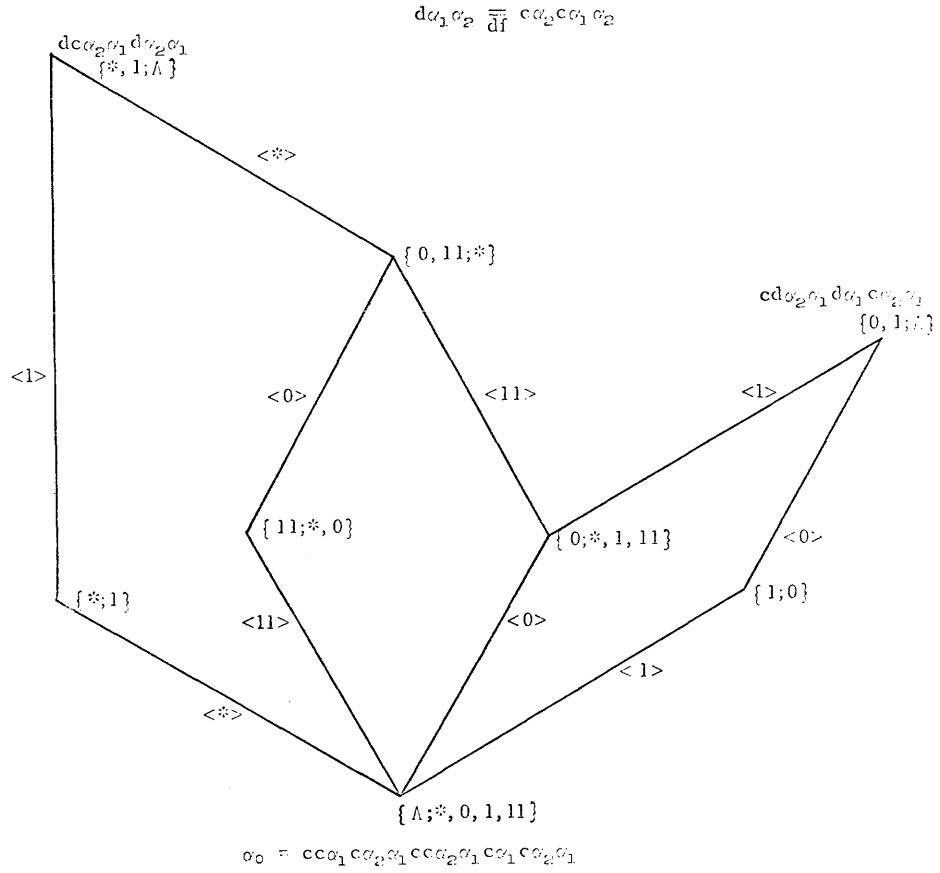


Figure 6—The Hasse diagram of E (α_0)

The address set is interpreted to be null if the multiplicity, m_i, is zero, i.e. if the definition is scope expanding at i.

Rule 3: If $*\in A_1$, then $<a>\ \{a \cdot A_1\} = \{a \cdot <*>\ A_1\} \cdot$

Rule 4: If $a \leqslant b$, then $<a>\{a,b\}=\{b\}$.

Rule 5: (the first commutation rule) If a and b are independent addresses (i.e. neither preceding the other), then $<a,b> = <b,a>$.

Rule 6: (the second commutation rule) If $m_i \neq 0$ (i.e. if the definition is not scope expanding at i), then $<a,a\cdot i\cdot a'> = <a\cdot a_{i1}\cdot a', \ldots, a\cdot_{mi}\cdot a\cdot,a>$. If the multiplicity is zero, $m_i=0$, then we can only say $<a,a\cdot i\cdot a'> \subseteq <a>$ (the functor $<a>$ has in its domain any expression with an occurrence of d at a; the functor $<a,a\cdot i\cdot a'>$ is more restricted because it also requires an occurrence at $a\cdot i\cdot a'$).

Rule 7: (the distribution rule) If the d-elimination functor F is applicable to the address sets A_1 and A_2, then it is applicable to $A_1 \cup A_2$ and

$$F(A_1 \cup A_2) = FA_1 \cup FA_2.$$

Suppose we now represent the distribution of occurrences of d and φ_d in an expression β by another expression of the form $\{C^{-1}d\beta;C^{-1}\varphi_d\beta\}$. Then, for the definition of figure 3, the expressions in $\rho\alpha$: $dc\alpha_2\ \alpha_1\ d\ \alpha_2\ \alpha_1$, $c\ d\ \alpha_2\ \alpha_1\ d\ \alpha_1\ c\ \alpha_2\ \alpha_1$, and $\alpha_0 = c\ c\ \alpha_1\ c\ \alpha_2\ \alpha_1$ $c\ c\ \alpha_2\ \alpha_1\ c\ \alpha_1\ c\ \alpha_2\ \alpha_1$, will be represented by $\{*,1;\Lambda\}$, $\{o,1;\Lambda\}$, and $\{\Lambda;*,0,1,11\}$ respectively. Moreover these three expssions all belong to $E(\alpha_0)$, consisting of eight expressions in a partial ordering whose Hasse diagram appears in Figure 6.

1. If β_1 and β_2 are forms over the extended alphabet and $\beta_1 \underset{d}{\equiv} \beta$, then there exists another form β_o such that $\beta \underset{d}{\geqslant}^{d}_{u} \beta_o$ and $\beta_2\ {}_{d}\beta_o$.

This is a very special case of the Church-Rosser theorem, which deals with the very general extension of formal systems by definitions. See, for example, Curry & Feys.[6]

2. Every expression β_1 over the extended alphabet determines a unique expression β_o over the original alphabet such that $\beta_1 \underset{d}{\geqslant} \beta_o$.

In other words, every equivalence class of expressions contains one and only one expression over the original alphabet, its unique 'normal form'.

3. For every expression β, $L(\beta)$ is a lattice. Figure 6 shows that this need not be true of $E(\beta)$.

4. $E(\beta)$ can only be infinite if the definition is scope expanding.

5. $E(\beta)$ can only fail to be a lattice if φ_d is self-dominating, and β contains a $\varphi_d\varphi_d$ -domino.

For example, this is the case in figure 6 because the occurrence of φ_d at 1 in α_0 dominates the occurrence

$d\alpha_1\alpha_2$	$\overline{\overline{df}}$	$c\alpha_2 c\alpha_1\alpha_2$
$d_1\alpha_1\alpha_2\alpha_3$	$\overline{\overline{df}}$	$c\alpha_1 c\alpha_2\alpha_3$
$d_2\alpha_1\alpha_2$	$\overline{\overline{df}}$	$d_1\alpha_1\alpha_2\alpha_1$
$d\alpha_1\alpha_2$	$\overline{\overline{df}}$	$d_2\alpha_2\alpha_1$

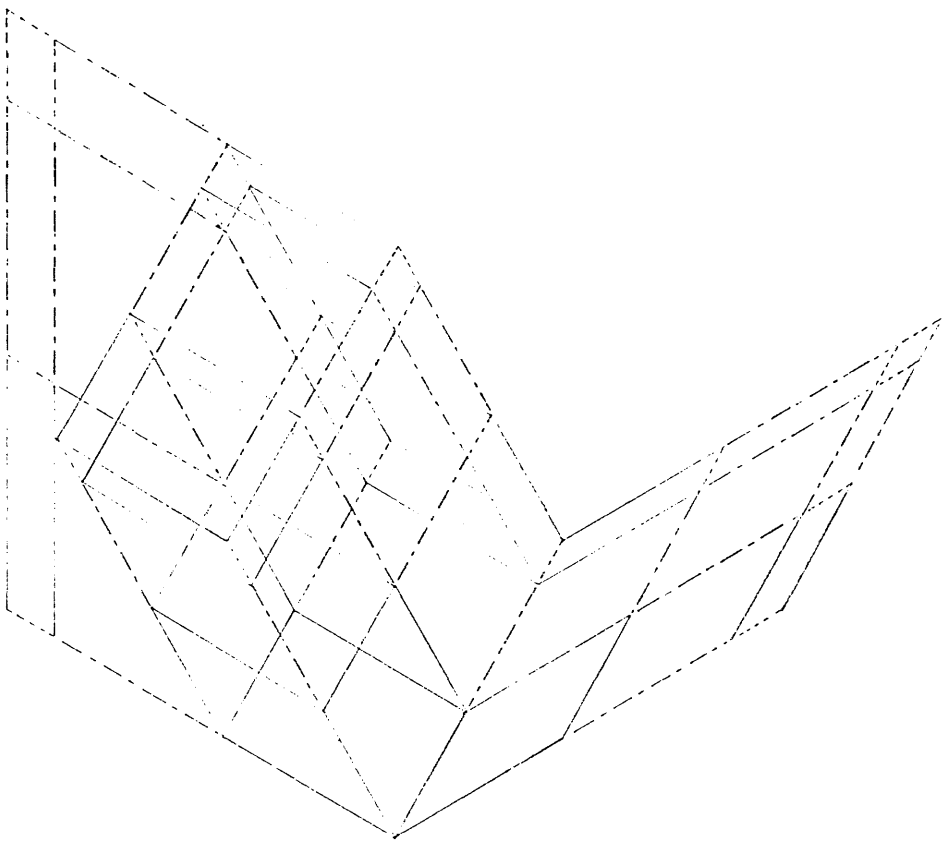


Figure 7—The Hasse diagram of E (α_0) when d is factored

at 11 and is dominated by the occurrence at *.

6. A partial ordering is called modular if, whenever $\beta_2 \underset{d}{\geqslant} \beta_1$, any two complete chains between β_2 and β_1 have the same length. Figure 6 is not modular because one complete chain between {*,1;Λ} and {Λ;*,0,1,11} is of length two and the other two are of length three. This happens because the definition is a breadth reducer, $C^{-1}\alpha_2\varphi_d = \{0,11\}$, and $C^{-1}\varphi_d\alpha_0 \supseteq \{*,0,11\}$.

7. If we factored this definition into a pure depth reducer, a pure breadth reducer, and a pure permuter, as mentioned before, we separate their several effects into three phases of extension, as one can observe by studying Figure 7.

CONCLUSION

One could extend the very special theory of this paper in a number of directions. For example, one could go on to study the effects of sets of definitions; the factorization into 'pure' types illustrated this. One could relax the 'completeness' condition we were using in our categories of languages, and consider various kinds of incomplete categories. One could relax the restrictions implied in our use of the word 'explicit' to permit definitions by cases, or suppression of variables, or even recursion. The fact that there is such a theorem as the Church-Rosser theorem means that some kind of light should emerge.

However, it seems to me that even our simple example has taught us some unexpected things.

First of all, the processing we have been discussing can occur at meta-language level as well as at object-language level.

For example, in Church's axiomatic system for the propositional calculus the first axiom is

$$\text{A1} \quad :\vdash \alpha_1 \supset (\alpha_2 \supset \alpha_1).$$

The Name A_1 belongs to the meta-language just as the punctuation, parentheses, and '| —: do. We can give A_1 several important types of interpretation in the meta-language, all of which we would like to use in the same system (see Gorn.[9] For example:

1. A_1 is an address in an addressing system for the storage of axioms and theorems.

2. A_1 is an instruction representing an operation on two expressions; $A_1\alpha_1\alpha_2$ will produce $\supset\alpha_2\supset\alpha_1\ \alpha_2$ in the language $\mathscr{P}\mathscr{R}$ of a category. This semantic effect carries with it all the syntactic effects of the explicit definition of figure 6 with A_1 for d and $\supset$ for c. In particular, the syntactic relation

$$A_1\supset\alpha_2\alpha_1A_1\alpha_2\alpha_1\equiv\supset A_1\alpha_2\alpha_1A_1\alpha_1\supset\alpha_2\alpha_1$$

would also have the semantic consequence that both expressions specify methods of deriving $\alpha_0 = \supset\supset\alpha_1\supset\alpha_2\alpha_1\supset\supset\alpha_2\alpha_1\supset\alpha_1\supset\alpha_2\alpha_1$. We also know, because of the A_1A_1 -domino in α_0, that there can be no expression in the $\mathscr{P}\mathscr{R}$ from which both can be derived.

This means that much of the processing needed in theorem proving is not essentially different, in spite of the shift in level of interpretation, from the processing at object level.

Moreover the introduction of explicitly defined symbols, as in $\supset\alpha_1\alpha_2 \overline{\overline{\text{df}}}\ V\sim\alpha_1\alpha_2$, and the introduction of systems of names of axioms and theorems are not only similar to each other but also play the same role in the meta-language as the introduction of new names of instructions, or of macro-instructions in a programming language. All these new symbols can also be interpreted as names of linguistic transformations on the object language, and therefore as operators in a meta-language over the object language.

This means that there is a second effect from our study. A number of seemingly different problem areas in information science, logic, and linguistics become identified:

a. The growth of a mechanical language by the explicit defining of new characters: it can be controlled if we have the definitional forms recorded, and use such processors as form recognizers, recognizers of the relation $\geqslant$ no matter what the 'd', reducers to normal form independent of d and φ_d, equivalence recognizers, etc. Each of these processors will permit one to vary the alphabets, the forms, and the definitions, of which there will be very many. But of the basic processors there will be only a handful, and they will be applicable at many levels as well as to many alphabets and many definitions. The system should also contain a number of the translators among the language functions of the category.

b. A number of problems in artificial intelligence: imagine that a language is specified for which a recognizer is either unknown or impossible; we want a 'heuristic' recognizer which will 'often' determine whether an expression belongs to the language by attempting to solve the problem of setting up a derivation for it. If the processor is halted before this happens, the recognition is left undecided. The investigations of Newell, Shaw and Simon, such as the logic theorist,[13] and the general problem solver[14] can be formulated this way, because the set of all 'true' expressions can be considered a language just as easily as the set of all expressions. This is because the axioms can be considered as ad hoc syntactic generators of good expressions, and such 'transformation rules' as modus ponens, substitution, etc. can be considered as derivation rules in a generative grammar.

The heuristic program recognizes patterns of applicability of these transformation rules in order to set up the 'subgoals'. The availability of our standard form recognizers, as efficient processors rather than heuristic ones, makes the main heuristics more efficient. See Boyer,[1] and Chroust.[4]

c. The extension of formal systems by definition: one considers the 'derivation rules' of the formal system to be the 'transformation rules of the language', in the sense of Carnap, just as described in b. This is why the material in the first four chapters in Curry & Feys,[6] leading up to the Church-Rosser theorem for general formal systems had its counterpart in our simplified and much more highly structured languages.

d. We handled an explicit definition as though it were an addition to the grammar of the language which permitted certain simple linguistic transformations to be performed. Our classification into depth reducers breadth reducers, etc. is a very primitive classification very much in the spirit of the much more complicated 'linguistic transformations' in the sense of Chomsky and Harris. Can we not consider that anyone who defines a new expression in a language is causing the language itself to change; is he not really changing the grammar of the language, and not merely adding new expressions?

e. Suppose we have two programming languages, such as, for example, the assembly languages of two different binary general purpose machines of the von Neumann type. We can consider that each instruction of either language is defined in terms of a common 'micro-language' of bit-manipulations. Some of these definitions are recursive, but many can be made explicit. Among the difficulties of machine-to-machine translation is the fact that many instructions, like 'ADD', do not really have the same definition in the different machines.

The same problem arises if a community of users, beginning with a common language, have the freedom

of introducing macro-definitions independently of one another. Sub-communities with common problems will develop different sub-languages by different systems of macro-definitions, and if these different sub-communitities do not remain in constant communication, their distinct 'dialects' will, in effect, grow into distinct languages, and the translations, even with only explicitly defined macro-instructions, will be come difficult because of what, in this paper, is called the domino effect among definitions.

The non-lattice effect we remarked on for definitions related by dominance we can now interpret as a danger of loss of communication; when the processors we have discussed are not available, the different sub-communities will not even know when they are saying the same things.

REFERENCES

1 M CHRISTINE BOYER
A tree structure machine for proving theorems
Moore School Master's Thesis August 1964

2 JOHN W CARR III
The growing machine
to be submitted to a computer publication

3 T E CHEATHAM
The introduction of definitional facilities into higher level programming languages
AFIPS Conference Proceedings Vol 29
Fall Joint Computer Conference 1966

4 GERHARD CHROUST
A heuristic derivation seeker for uniform prefix languages
Moore School Master's Thesis August 1965

5 *A CHURCH*
Introduction to mathematical logic
Vol I Princeton 1956

6 H B CURRY R FEYS
Combinatory logic
Vol I North Holland 1958

7 B A GALLER A J PERLIS
A proposal for definitions in ALGOL
to appear in Communications of ACM

8 SAUL GORN
Common Programming Language Task
Rep AD 236-997 July 1959 and
Rep AD 248-110 June 30 1960 U S Army
Signal Res and Develop Labs Fort Monmouth
N J Contract No DA-36-039-SC-75047

9 IBID
The treatment of ambiguity and paradox in mechanical languages
Am Math Soc Proc Symposia in Pure Mathematics
Vol V (1962) *Recursive function theory*
Apr 1961

10 IBID
Mechanical pragmatics: a time motion study of a miniature mechanical linguistic system
No 12 Vol 5 December 1962 pp 576 589

11 IBID
Language naming languages in prefix form
Formal language description languages for computer programming
North Holland 1966 pp 249-265

12 IBID
An axiomatic approach to prefix languages
Proceedings of the symposium at the international computation centre
Symbolic languages in data processing
Gordon and Breach 1962

13 A NEWELL J C SHAW H A SIMON
Emperical explorations of the Logic theory machine: a case study in heuristics
Proceedings of the western joint computer conference 1957 pp 218-230

14 IBID
Report on a general problem-solving program
Information processing
UNESCO Paris 1959 pp 256-264

15 T J OSTRAND
An expanding computer operating system
Moore School Master's Thesis December 1966

An optical peripheral memory system

by R. L. LIBBY, R. S. MARCUS* and L. B. STALLARD
Itek Corporation
Lexington, Massachusetts

INTRODUCTION

The increasing extension of computer technology to applications involving the accumulation and use of large libraries of digitized data (10^{12} bits or greater) has stimulated interest in new digital peripheral memory techniques.[1,2,3]

Of the many characteristics and performance parameters of interest related to peripheral memories, achievable digital storage density is a fundamental parameter since it contributes strongly (but not uniquely) to the access time, data transfer rate, cost per bit, and other important properties of digital peripheral memories.

The particular peripheral memory described in this report uses optical techniques for recording and reading digital data. The search logic and general purpose computer associated with this memory have allowed practical demonstration of the recording and reading of digitized information and the functional use of the peripheral memory as a content addressable store.

The use of optical techniques in the subject memory demonstrated that many memory performance characteristics typical of magnetic technology can be equalled or exceeded.

It would appear, however, that the promise of optical techniques in the peripheral memory area lies in the area of storage and retrieval of digitized data in capacities of 10^{11} or greater for either on-line or off-line storage or retrieval applications. This paper describes first, an operational prototype system and then presents technical considerations for achieving mass digital storage capabilities.

The system for evaluation of the photo-optical memory technology

A complete computer system was designed around the photo-optical memory and tailored for combined algorithmic (programmable), decision table, and tag-associative memory type of processing. This combination of processing appears to be typical of a broad class of non-numeric information handling problems that range from language translation to image pattern analysis and recognition and proved to be a more than adequate test vehicle for assessing the memory technology.

Figure 1 is a picture of the final operating test system. The photo-optical disc reading unit occupies the two-bay cabinet to the left of the chain printer and magnetic tape transport units. The memory search logic is incorporated in the three-bay unit in the center background, along with the general purpose computer logic.

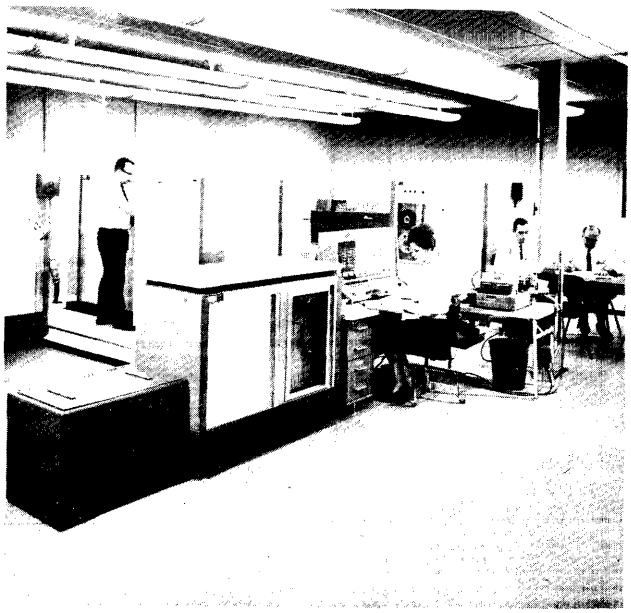


Figure 1—Overall view of memory test system

Figure 2 shows the Disc Writer Unit which is fully automatic except for loading and unloading the photographic disc.

An overall block diagram of the test system environment for the memory is shown in Figure 3.

*Currently with the Electronic Systems Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts.

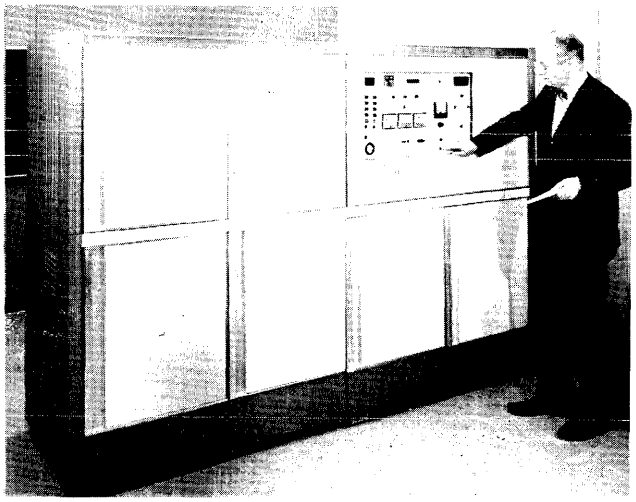


Figure 2—Disc writer unit

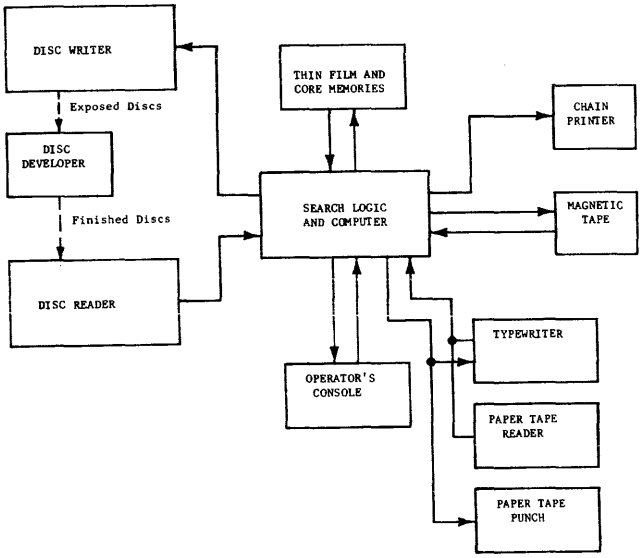


Figure 3—Test system block diagram

The magnetic tape unit is used primarily for the input of source data to be recorded in the photographic unit record, or photostore disc, of the system.

The control computer buffers each track worth (9,600 seven bit characters) of source data during the memory writing operation to insure affirmative parity checks before transferring the data to the Disc Writer Unit. The data transfer and writing occurs at a rate of 100,000 bits per second. The Disc Writer records these data serially by bit on each concentric track of the disc. Disc writing can terminate after any of the 2350 total possible tracks are completed but not during the writing of a track. The exposed (recorded) disc is then removed (manually) from the Writer and placed in a Developer Unit. After chemical development, the recorded disc is either stored for later use or manually placed in the Disc Reader Unit for use in retrieval or dictionary look-up processing. Mounting of the disc in the Disc Reader is as simple as mounting a magnetic tape reel. Once mounted, the photostore disc provides an on-line unit record of 150 million bits which can allow access to randomly located entries within it, in an average time of 15 milliseconds. When a desired entry is found, read-out into the computer memory occurs at a 4 megabit rate.

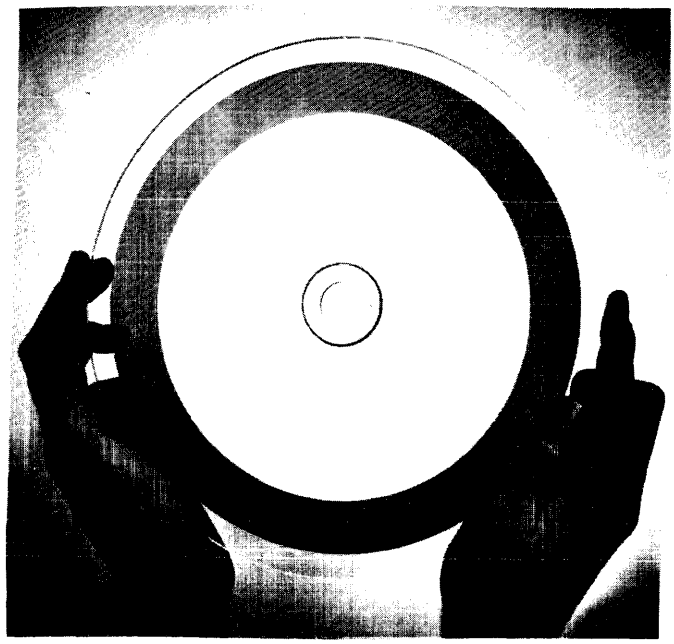


Figure 4—Photographic memory disc

The photostore disc

The short random access time (15 milliseconds), high read out rates (approximately 4 megabits per second), and high unit record capacity (150 million information bits) achieved by the single reading station, results from the use of optical techniques for data writing and a photographic film emulsion as the storage medium. A glass disc, one-quarter inch thick, and 10.5 inches in diameter is coated with Eastman Kodak Type 649F high resolution emulsion. This is an emulsion formerly used primarily by spectroscopists but recently a favorite in holographic work.

The information is recorded in a one-inch annular region (centered at a 4.25 inch radius), in the form of 2,350 concentric tracks (see Figures 4 and 5). Each track of information shares a transparent border and an opaque border with its neighboring tracks (see Figure 6). The stored information is recorded serially by bit on each track and 67,000 bits are recorded on each track, at a density on the inside track of approximately 2,850 bits per inch. The code used (Man-

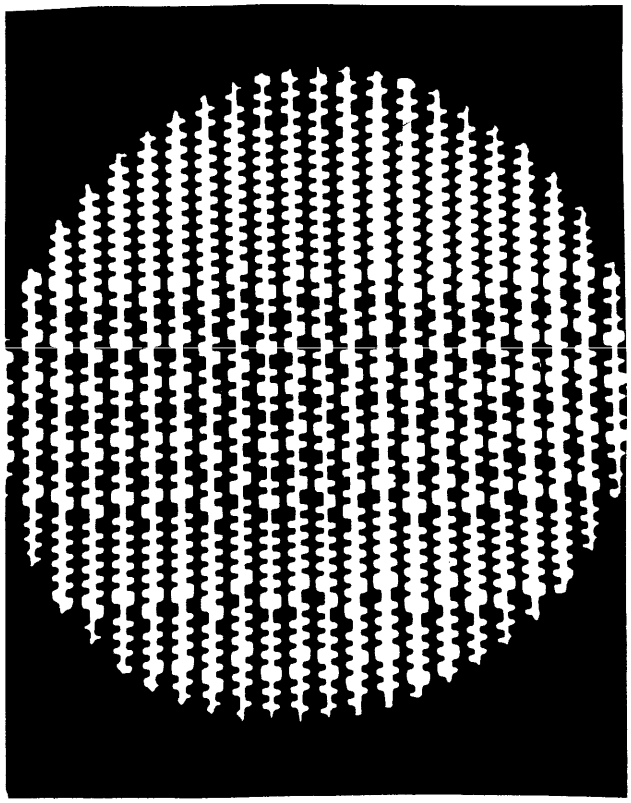


Figure 5 – Microphotograph of disc information tracks

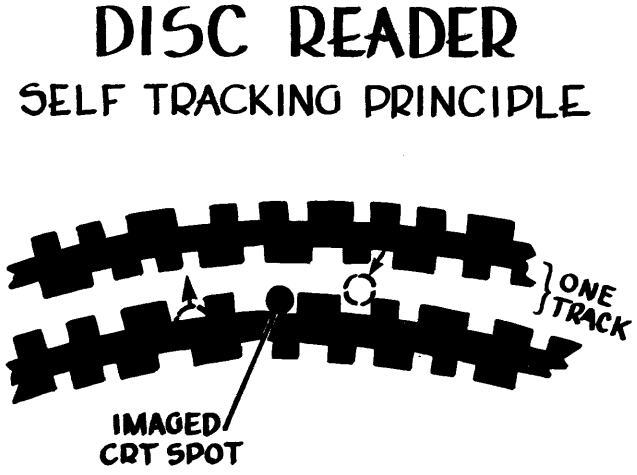


Figure 6 – Disc reader self tracking principle

chester), provides a short-time average light transmission that is independent of the information stored. For example, each logical bit value recorded as an opaque or clear mark is followed immediately by its complement. Thus a string of logical zeroes consists of an alternating sequence of opaque and transparent 5 micron square marks. The all zero code, 6 logical zero bits plus 1 odd parity bit per character, is reserved for use as an operational code recognized by circuitry. Thus, the sequence of bit marks representing the double character sequence ($00_8 01_8$) is uniquely available as a detector synchronizing code. The octal zero "shift" code, coupled with the other characters provides 62 other possible double byte code words which, with appropriate hardware, could be used as special hardware recognizable operational codes or data delimiters. The Disc Memory currently uses the following five combinations:

Character code	Function	Mnemonic	Comment
$00_8 01_8$	Detector Synchronizing and Entry Delimiting	$\alpha\beta$	Machine Detected
$00_8 02_8$	Argument-Function Separator	$\alpha\tau$	Machine Detected
$00_8 03_8$	Substitutes for 00_8 in stored information	$\alpha\gamma$	Machine altered to 00_8 upon read out
$00_8 04_8$	Delimiter introducing Track Identification No.	$\alpha\delta$	Machine Detected
$00_8 77_8$	Match Anything Code	$\alpha\nu$	Machine Detected

There is nothing fundamental restricting the use of less dense codes in systems of this type, such as 8, 9, 10 bit, etc. code words, if one finds that either the reserving of 00_8 from available data-representing code words is too restrictive or the transforming of 00_8 into a double byte code prior to storage is cumbersome.

As noted above, the system uses a two-character code ($00_8 01_8$) to delimit or bound the data entries in the memory, and another two-character code ($00_8 02_8$) as an internal separator between the argument (look-up tag) and the function (read out) portion of an entry on the storage disc. Entries may contain one information character or up to 9,600.

Photostore disc reading

In utilizing memories having on the order of 10 million information marks per square inch, one cannot rely on absolute mechanical positioning of the read-out mechanisms. In the optical memory reader,

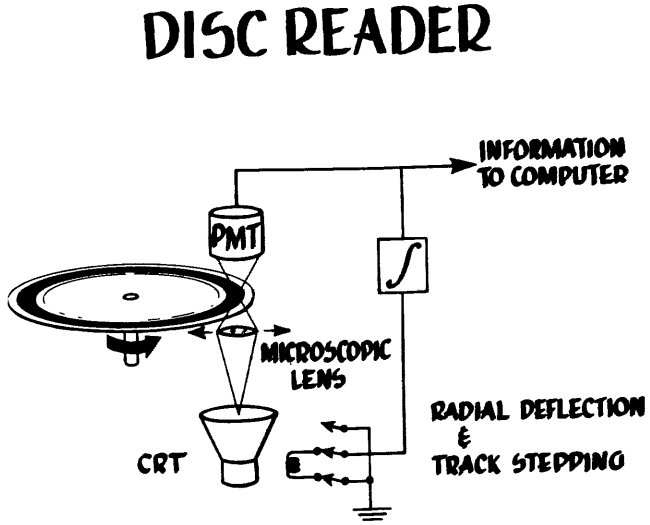


Figure 7

a reduced optical image of a Cathode Ray Tube (5AUP24) spot is positioned on the emulsion containing the digital information mark pattern. As depicted in Figure 6, the imaged CRT spot may wander from the track locus, as the photostore disc rotates at 3600 rpm. The generalized closed loop servomechanism depicted in Figure 7 keeps the image of the read-out spot centered on the information marks by feed-back control of the radial deflection of the CRT.

Since the "on track" condition is stable, track stepping is accomplished by reversing the sense of deflection on the CRT and simutaneously "nudging" the spot in the radial direction desired (i.e., toward the disc memory center, or toward its outside edge). To make the tracking mechanism action independent of the stored information, the Manchester code is used which provides an average "on track" light transmission of 50%. The servo response is limited by the 5.4 microsecond, short term, circuit-integrated average of the output of the photomultiplier tube (the PMT that reads the information-modulated CRT light). A non-integrated version of this output goes to the search logic and control computer. Stepping of the reading spot is controlled either by computer instructions or the automatic search logic. Total "many-track" stepping time (averaged per track), including instruction execution and servo settling time, is estimated at 7 microseconds.

Since track stepping cannot proceed at a rate approaching the rate of occurrence of information marks (when "on track") and it is desirable to assure time for the reading spot to "lock on" each track that it passes, the radial velocity of the read-out spot is set so that radial search time corresponds to an equivalent velocity of about 61 inches per second. Programmed track search and entry sampling reduces this to 50 inches per second. The tangential motion of the information tracks relative to the read-out spot (by disc rotation) is approximately 1700 inches per second.

Disc format optimization

It can be readily shown that the disc format has about 80% of maximum capacity for the given mean access time. The ratio of the two average coordinate search velocities (random search) is about 80% of the ratio of their corresponding coordinate dimensions. That is:

$$\overline{V}\text{ tangential} \times 2 : \overline{V}\text{ radial} \times 3 = 22.7$$
$$\Pi R : W = 28.3$$

and

$$2\Pi R : W = 28.3$$

where:

W = width of information annulus = 1 inch

R = average radius of information annulus ≐ 4.5 inches

$\overline{V}$ radial = 50 inches/second

$\overline{V}$ tangential = 1700 inches/second

The factors 2 and 3 correspond to cyclic and bilateral search motions in the tangential and radial directions, respectively.

and:

$$(2 \times \overline{V}\text{ tangential}) \div (3 \times \overline{V}\text{ radial}) : 2\Pi R/W \approx 0.8$$

The disc memory unit functions

The two-bay Disc Memory Reader Unit contains five major functional sub-assemblies and utilizes seven control logic lines between it and the search and control logic. These lines pertain to track step commands, error flags, clocking, and entry delimiter signals. The five major functional sub-assemblies are:

1. CRT Light Source Generation: The 5AUP24 CRT is run at a beam power of 1.6 watts with a 0.012 inch diameter spot. Since this is a phosphor temperature-limited operation, the spot is kept moving on the phosphor in a 2.5 inch diameter circle at 100 cycles per second. Protection against loss of deflection is provided. A reference PMT continuously views the CRT and maintains constant CRT spot light output.

2. Lens Motor Assembly: The microscope lens used to image the CRT spot is mounted on a "voice coil" which is supported as an air bearing on the center pole piece of a large cylindrical electromagnet. The one inch radial motion of the field of view of the lens, which statically encompasses about 100 data tracks, has the necessary acceleration such that it does not limit the track stepping time as determined by the tracking servo settling time.

This lens motor is injected with a properly phased component of the 100 cycle signal that is used to generate the CRT spot circular motion, in order to cancel any radial motion of the spot's image on the disc. The 100 cycle tangential motion (along the information tracks) is tolerated.

3. Tracking Servomechanism: Both the CRT radial deflection (radial with respect to the disc spindle center) and the lens motor (lens position) are driven by the amplified error signal generated as the integrated PMT output (see Figure 6) deviates from a "grey level" reference value. The lens motor corrects for all perturbations up to about 120 Hertz and the CRT radial deflection for perturbations having frequency components from that cut-off point up to 150 to 200K Hertz which is well below the energy band of the information being read out.

4. Signal Detection and Processing: The analog signal output from the disc reader PMT is shaped by a delay line correlator and a detector clock phasing signal is derived from the information mark transitions (clear to opaque, etc.).

Since the disc output signal contains two pulses of opposite polarity for each bit recorded on the disc, this fact is utilized by incorporation of two detectors whose detection sampling intervals are shifted in time by one information mark time interval. Detection of a parity error in say the "true" detector automatically causes selection and use of the character in the "complement" detector. Circuit means are provided to prevent improper phasing of these detectors.

5. Air System: Approximately one-half of the Disc Reader volume is occupied by an air compressor and de-humidifier provided for the lens motor air bearing. Experience has indicated that the de-humidifier unit can probably be eliminated and the compressor and air reservoir system reduced in size.

Photostore disc memory searching

Finding an entry in the photostore memory to begin read out can be accomplished in two different modes:

a. Content Addressing (track search followed by entry search)
b. Vicinity Search (absolute track addressing) followed by Content Addressing

In the Content Addressing Mode, a string of characters (this can be several thousand characters long) is placed in the Look-Up region of the memory of an associated computer, or if the system is designed for an entry of fixed length, it can be inserted in an interface register and the Look-Up instruction executed. As soon as the CRT spot image, which may be on any track of the photostore disc, encounters an entry delimiter ($00_8 01_8$) then each succeeding character is matched sequentially with the corresponding sequential character in the Look-Up memory region (or register). The moment an inequality of character code match is encountered, the reading spot steps up or down one track on the disc, based on the sign of the inequality. The matching of characters in an entry argument with those in Look-Up region begins over again the moment an ($00_8 01_8$) is again encountered. Since the arguments of the entries in the dictionary are arranged in order by their code value (as left justified fractions, with each character value being a digit of the fraction) then this step track, attempt a sample match, and then step track sequence continues until the sequence of track-sample matches produces a "memory sample less than" to a "memory sample greater than" transition. It will be noted that during this "Track Search" phase, even an extra match of an entry argument with the sequence of characters being looked up, will not cause read out of an entry's function. This prevents a match of a shorter (hence lower valued) memory-entry argument with a short sequence of symbols being looked up, when a longer (higher valued sequence) may exist in the photo-disc memory.

After the "less than" "greater than" track sampling sequence criterion is satisfied, the "Entry Search" phase of the Look-Up instruction occurs. The read-out spot stays on a track during the Entry Search phase until either an argument is found that matches every character in the look up region as far as its intra-entry delimiter codes, or until an "end of track" code word is encountered. In the latter case, the read-out spot jumps to the next track containing lower argument code values and continues entry search, and so forth, until a matching argument is found or end of the memory is reached. When a matching argument is found, all parts of the entry following the intra-entry boundary between argument and function and the ($00_8 01_8$) function terminating code word are read out into a memory field (staticizer region) of the attached computer.

The value of this "longest match", content addressable, search procedure in language processing has been described in some detail.[4] The sampling of one entry on each track in order to make the "sampled comparison" can slow the content addressable search procedure if the argument (or tag) of entries are long. Accordingly, a means has been provided in this system to perform a programmed "vicinity search" phase prior to entering the "Track Search Phase" of the Look-Up instruction. This is accomplished by automatically reading from a register the track number of any track that the read-out spot is resting upon (by employing a "track identify" instruction). These track numbers may be recorded as frequently as desired on a memory track or as little as once at the beginning of each memory track. Preceding a look up instruction, this track number is ascertained and compared with the approximate track location of the arguments having the first character value of the first character in the look-up region (this is determined from a small table in the control computer's memory). The program then gives a command to step the necessary number of tracks in the proper direction prior to giving the look-up instruction.

Absolute Track Addressing is accomplished by use of the track number identification and programmed track step commands.

Disc memory access time and capacity relationship

The average timing of the accessing of randomly located records on the photostore disc can be approximately expressed in a quantitative manner. A general relationship can be derived, relating the size of a dictionary that is stored in the photo-disc memory and the average random access time to an entry:

Relationship of total dictionary size and look-up time:

D = dictionary size in bits
E = dictionary size in terms of the number of entries
B = average number of bits per entry, including delimiters (assume function and argument are equally long)

Assumptions and fixed parameters:

(a) Servo settling time and step-track instruction execution in track stepping is approximately 7 microseconds
(b) $D = B \times E$
(c) T = average look-up time for each of a large number of entries randomly located in memory
(d) K = redundancy (repetition) of entries (=1, 2, 3...etc.) in terms of repeated sectors on a track
(e) C_e = capacity of photostore disc that is effectively used; $C_e = K \times D$ bits
(f) N_T = number of tracks utilized on disc, where:

$$N_T \approx \frac{C_e}{7 \times 10^4} \text{ for } 10 << N_T < 2351$$

$$\text{or } N_T \approx \frac{K \times D}{7 \times 10^4} \text{ tracks}$$

(g) memory serial scan bit rate (along track) 4×10^6 bits per second

If vicinity search is used, the sum of the following terms will equal the access time in seconds:

Term 1: Radial Track stepping time $- 7/3 \times N_T \times 10^{-6}$ seconds
Term 2: Rotational latency time $- 1/2 \times 1/K \times 16.7 \times 10^{-3}$ seconds
Term 3: Vicinity track search; programmed file search of a 64 entry table $-(64/2) \times 30$ cycles $\times 0.8 \times 10^{-6}$ seconds $= 0.77 \times 10^{-3}$ seconds
Note: The control computer has a 0.8×10^{-6} second cycle time
Term 4: Final track stepping and entry sampling within 20 tracks of desired track $- 20 (B/(2 \times 4 \times 10^6) + 7 \times 10^{-6}) = 0.40 \times 10^{-3}$ seconds for B = 100
Term 5: Function read-out time $- B/(2 \times 4 \times 10^6) = 0.013 \times 10^{-3}$ seconds

Since terms 3, 4, and 5 contribute about 1.2 millisecond, the access time (for entries where B ≈ 100 bits) can be approximated by the following expression:

$$T \approx 2.3 \times 10^{-3} N_T + 8.4 \div K + 1.2 \text{(milliseconds)}$$

where, again:

N_T = total number of tracks of information on photo memory disc
K = number of repeated information sectors per track

The number of tracks (N_T) can be expressed in terms of the number of entries in the photo memory. The expression for the average access time to randomly located entries becomes:

$$T \approx 0.033 \times 10^{-6} K \times B \times E + \frac{8.4}{K} + 1.2 \text{(milliseconds)}$$

It should be noted that the last term of the above access time expression is not very sensitive to the length of an entry (B) for a given capacity. A ten-fold increase in B (to 1000 bits per entry) would change the final constant in the above expressions from 1.2 milliseconds to 2.8 milliseconds.

Photostore disc memory writing

The use of non-erasable photo-optical phenomena as a digital information storage method causes more conceptual than actual operational difficulties. It is to be noted that in many magnetic tape oriented systems, erasibility features are used primarily to recover the use of the storage medium rather than as an updating feature. The development program for this system has shown that writing rates competitive with 90 kilocharacter per second magnetic tape transports would be easily achieved with long-life, medium power, C-W lasers. The throw away costs of optical recording materials, if obsoleted by re-written records, could be negligible compared to the processing costs attendant to any large record updating task, using say a re-usable medium.

The photo-memory writing process starts with a prepared batch of data on IBM-format compatible magnetic tape. This data has been segmented approximately into 9,600-character blocks, since this is the capacity of each of the 2350 tracks of a photo-disc. The control computer memory writing program reads one block of information into the core memory and turns on a "Ready to Write" light on the control

panel of the Disc Writer unit. An operator has previously mounted an unexposed disc on a spindle in the Writer and he presses the "Start-Write" button when the computer signals "Ready to Write". The block of information is transferred under program control, six characters at a time, to the Writer Unit. The latter unit, serially by bit, by means of an electro-optical modulator, modulates a 25 milliwatt He-Ne C-W laser beam with the information. The laser source beam has been split into two beams (Figure 8). One of these, modulated by the information bits to be recorded, is imaged as a 1.2 by 5 micron slit image onto the disc emulsion. The motion of the emulsion of the moving disc with respect to the slit-like image and the timing of the on-off sequencing of the electro-optical modulator produces a 5 micron square clear (unexposed) and a 5 micron square opaque (exposed) sequence of areas on the disc for each logical "zero" to be recorded. The opposite sequence of an opaque and a clear area is used to record a logical "one". The second portion of the beam forms a slit-like image (the border image) which is left "on" continually while writing information marks. By switching the position of this border image every time a new track is written, alternate clear and opaque track borders

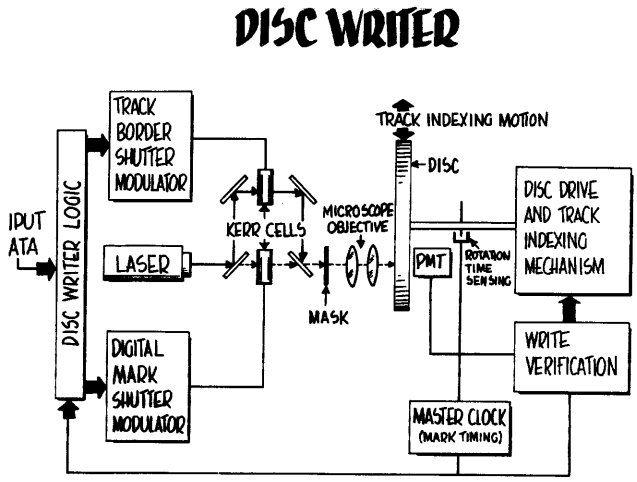


Figure 8 – Disc writer diagram

are obtained as illustrated in Figure 5. When one track of information is written (9,600 characters and one spindle revolution), the spindle containing the disc is electromechanically indexed radially about 430 microinches (11 microns). During this indexing, the control computer is reading the next block of source magnetic tape. At the start of the next revolution of the disc spindle, the foregoing sequence is repeated. Disc spindle rotation and modulator timing are derived from stable frequency sources and it has been found that one or two characters of buffering (empty of significance) suffices to take care of the overlapping of the beginning and end of a written track. All track beginnings are approximately aligned by means of a synchronizing magnetic pick-up on the writer spindle but radial correlation of track to track information at the bit level does not otherwise exist intentionally.

The Disc Writer writes at instantaneous rates of 100,000 bits per second, but because of the interlaced track indexing motion (spindle radial indexing), the effective writing rate is 50,000 bits per second or about 7 kilocharacters (6 bits plus parity) per second. A design for interruptible spiral track writing has been completed which avoids this loss in effective writing rate. A more detailed technical description of the photo-optical aspects of the Disc Writer has been presented.[5]

When completion of the disc (or partial disc) is indicated (by track number), the exposed disc is removed manually and placed in an automatic developer unit (Figure 9). A standard photochemical development is carried out by this unit with the precaution that all solutions are filtered (to 1 micron) immediately before their application. A completed dry disc, with archival quality, is obtained in about 15 minutes. This process probably could be reduced, by using heated and special solutions, to a period of 3 minutes.

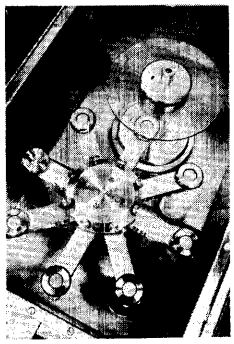


Figure 9 – Automatic disc developer unit

The search logic and photo memory related instructions

Since dictionary type look-up operations were expected to make up an important segment of the memory test operations, a special search logic comparator was added to the Disc Reader that allows a set of single instructions or computer-commands to address the 150 million bit photostore memory on a content addressable basis. The following instructions facilitate search operations in the large memory:

Look up

This instruction locates the entry whose argument (TAG) makes the longest exact match with the indefinitely long string of characters stored in fast memory beginning at a specified location. When such

an entry is found in the disc (argument match) its function (this can be either data or program) is automatically read out into a designatable (settable by manual switches) memory field (staticizer region) of the control computer's memory.

Retrieve

This instruction was devised to search the large memory for a pattern of symbols of variable length and of uncertain character content. An octal code that is to be recognized as a "mask" character is manually set into a switch register. Using the Greek letter "ω" for this value, and the letter "x" for any alphanumeric character in the memory, and "x_N" being a particular specified alphanumeric character, then all entries in the memory having a tag—or argument—such as $x_1x_2xxx_3x_4$ or $x_1x_2xxx_3x_4x$ would be retrieved by applying the RETRIEVE instruction to an input character sequence such as $x_1x_2\omega\omega x_3x_4\omega$. This device can be useful in circumventing common spelling errors in the input data to be processed and for masking certain portions of character patterns that are being looked up in information retrieval applications. It is especially useful in applications where it is desired to retrieve information on a number of different keys but where it is not practical to presort the information on each key individually. Read out of matched information is accomplished into a staticizer region as in the LOOK-UP case. This instruction starts its matching memory scan at some specified argument and it terminates when reaching the end of the dictionary or when the staticizer region is filled.

Dump instruction

The inclusion of this instruction recognizes that it may frequently be desirable to bring sections consisting of many contiguous total entries of the photo-optical memory contents into the computer's memory where programmed search (or scanning) of total contents may be accomplished. The instruction operates by first finding a matching argument of an entry for a specified beginning point (entry) then it provides a complete sequential read-out (arguments as well as functions) of the photo-optical memory in a decreasing argument value direction until one of the special superboundary codes that have been previously inserted in the dictionary is encountered.

Masking features

The three basic content addressable memory search instructions, described above, constitute the table or dictionary look-up repertory of the system. An additional code masking feature is important since it allows both economy and flexibility in the dictionary make up. This masking feature uses a particular octal code called the "match on anything" character. It is used in the dictionary only, and it is equivalent to the lowest valued character in the disc memory (that is, it is the last character value on which a match is attempted during Look-Up instruction execution). In a sense, it can be termed a "fail safe" matching character.

Identify track

This is a specific command that allows query by the control computer of a special register which contains the track number identification of the track currently being read. Since track identity codes can be arbitrarily placed around a data track, latency time for track identity can be arbitrarily reduced. The operational code delimiting track identity numbers is uniquely recognizable (separably from recorded data) and this allows the updating of this register to be automatic if this is desired.

Track stepping instructions

Two commands are provided in the search logic for stepping the read-out position of the Disc Reader light spot image. These are the UP TRACK and DOWN TRACK commands that cause the read-out to occur on data tracks that are adjacent to the one currently being read. These commands are capable of being automatically sequenced and controlled.

Content addressable search logic improvements

The programmers who have programmed the system have made good use of its distinctive operational features. The experience has, however, "whet appetites" for inclusion of additional memory search features. The following list of "improvements" has been suggested for the disc search logic.

1. Allow the value of the masking code (ω) to be program settable. This would allow greater flexibility with respect to masking the variety of input code sets that can be handled by the system.
2. Accomplish termination of the RETRIEVE instruction by hardware determination of the fact that no further search matches are possible. This would save time in the completion of the RETRIEVAL execution.
3. Achieve automatic read-out and conditional storage (depending on subsequent match or no match) of the argument. In many applications this would save repetition of the entry argument in the function of the entry. In cases where match occurs on a ν (match anything character) in an argument, the characters involved can be determined by program.

4. Introduce a program controlled "Count Match" so that the programmer can set threshholds for the number of bit or character matches required in a search (e.g., majority match). This would be useful in pattern recognition operations.
5. Provide a capability for program setting of several search logic registers such that certain operational codes (derived from the "shift" code) can cause skipping of the search match operation across "sub-fields" of either entry arguments or of the input data stream. This would be useful for implementing syntactic analysis rules and multifaceted record retrieval. This feature would additionally allow numerically ordered super-entries to be incorporated in unordered memory contents. Such a procedure would allow use of the RETRIEVE operation (serial search) with a key word which would then yield a sentence number permitting normal dictionary (random) look-up of the pertinent sentence(s).

The general use of the content addressable features of the optical memory in association with a general purpose computer tailored for non-numerical information processing is described by the authors in a separate paper.[6]

The potential of optical memory techniques

Certain generalizations concerning the performance potential of optical techniques for the storage of digital information can be drawn from the experience of the authors and from work recently reported in the literature. These generalizations cover the three important aspects of a digital peripheral memory, namely (1) Realizable maximum storage density, (2) Storage media and (3) Storage formats.

Realizable maximum storage density

The three spatial dimensions and the light transmission variable (or density) of optical media can be used in a variety of quantitative combinations for the storage (and retrieval) of digital information. Two well known extremes are the "discrete volume per bit" and the "distributed volume per bit" techniques characterized by the technique described in this paper and by holography, respectively. In the discrete volume case which is discussed here, the maximum information capacity per unit volume of recording material is determined by the minimum size of elemental recording volume that can be achieved. This is in turn determined by the numerical aperture (or F number) of either the recording or read-out lens.

If the memory system involves rapid relative motion of the recording medium and the recording (or reading) optical system then consideration must be given not only to the size of the optically recorded image element but also to the positioning of it with respect to other recorded elements and with respect to the read-out optical system as well. A lens having a numerical aperture of 0.5 (or an F-number of 1.0) was used in the Disc Writer described in this report. The "depth" of a "point" image for this lens is about 2 microns (one micron = one micrometer = one millionth of a meter), and tracking (or clamping) of read-write optical planes with respect to some surface of the recording material must be maintained to at least this degree, thus increasing the depth dimension of the elemental volume dedicated to a bit to say 4 microns. Similarly, the cross sectional area of the image is about 1.2 microns in diameter. Here again, an additional 1.2 microns should be reasonably dedicated to allow for physical tracking requirements. The elemental volume of recording material that is dedicated to the recording of a bit of information (in binary density form) now becomes, with these assumptions, about 18 cubic microns. If a recording surface layer 4 microns thick were used with these criteria then a reasonable maximum of 150 million bits per square inch of recording material could be stored and read out.

Photo-optical phenomena, and possibly thermo-optical phenomena, offer the possibility that the optical density of each elemental bit volume (in the discrete volume per bit mode) can be quantized to a greater extent than a binary mode. While this is true, the tolerance on spatial tracking and read-write energy levels may tighten to a degree that very little is gained in an overall system sense. Holographic or diffraction grating techniques represent a different trade-off of the roles played by the variables of an optical system. The density variable is used to an extent that quantizing levels are determined by the area average "noise" of the recording medium and inter-symbol noise, and a bit volume is distributed over a larger part of the recording medium. The relative advantages of these two techniques may well be determined by system implementation factors rather than achievable storage densities.

Storage media

Success in the use of thermal-optical effects in the recording of information has been reported[1,7,8] and a provocative proposal for optical writing in a magneto-optical memory has been published.[9] Since the early publication concerning photo-optical techniques for information storage,[10] tremendous improvements in optical energy generation, and in flexible media coating and manufacture, have been made.

If a recording medium or base coating of suitable cost, uniformity, physical-chemical and general environmental invulnerability is found and proven then the thermal-optical memory phenomenon offers attractive system attributes. The single step recording process (no latent image development) and the simultaneous read out during recording offer both reassurance and operational simplification to the user. Table I presents a summary of some reported energies that have been used (or proposed) to record digital information. Relative recording speeds with identical writing source energy (and optics) are also indicated.

TABLE I
OPTICAL MEMORY WRITING ENERGY

Method	Material	Approximate Ergs per Mark*	Relative Writing Rate	Source
Photo-optical (visible red)	Spectroscopic Film Type E.K. 649F	10^{-5}	3×10^{5}	1
Thermal-optical (visible blue-green)	Opaque coating on Mylar base (UNICON)	1	3	2
Thermal-optical (visible red)	Evaporated metal coating on glass 0.05 micron thick (lead,tantalum)	0.1	30	3
Thermal-optical (visible red)	Dyed plastic coating on glass - one micron thick	0.2	15	3
Thermal-optical (visible red)	Developed aerial photographic film			
	High Density	0.3	10	4
	Low Density	3	1**	
Magneto-optical (Ultra-violet)	Gadolinium-Iron-Garnet (theoretical)	3	10^{-3}	5

*Based upon a one square micron information Mark.
**Normalized at 100,000 marks/second for 30 milliwatt visible light energy incident on one square micron mark area.
SOURCES
1. Authors
2. Becker[1]
3. Carlson et al[8]
4. Chitayat[7]
5. Forlani et al[9]

Storage formats

At the density of recording that is achieved in the system described by this paper (6 million bits per square inch) about 1000 square feet of a recording surface would be required to hold a trillion bits (10^{12}) of information. If 150 million bits per square inch represents a practical limit for discrete area optical memories then such a memory would require about 50 square feet of recordable surface. There appears to be general consensus that libraries of digitized information will continue to grow as will the need for more economical and higher capacity peripheral digital memory modules offering both automatic and manual accessibility. If optical techniques are to play their expected role in satisfying this need, consideration must be given to both the unit record and storage module format for the record material. A review of the physical forms that magnetic technology has employed (drums, discs, strips, loops, cards, reels, etc.) provide little guidance since erasability and re-usability features of magnetic technology confuse possible parallelisms. The use of low volume flexible media in either reel or some segmented form is indicated, however.

Flexible media are readily adaptable to the handling required by optical systems. For example, the authors have conceptually designed a mass memory system employing both flexible disc and strip forms of unit record. Figure 10 shows a view of an experimental reading lens holder that has been retracted just prior to radial withdrawal, to allow automatic disc changing. This lens holder is also an air bearing clamp that holds the moving recorded emulsion surface (on a flexible medium such as disc or strip) at the focal plane of the reading optics. Mass on-line storage providing random access to 10^{12} bits in the one to ten second access time range can be achieved by "juke box" techniques for discs or self-threading cartridge changing for reels. Since much of the mechanical transport technology developed for magnetics is applicable to optical memory media, progress in productizing trillion bit (or greater) optical memories would appear to be highly dependent on the nature and magnitude of user requirements.

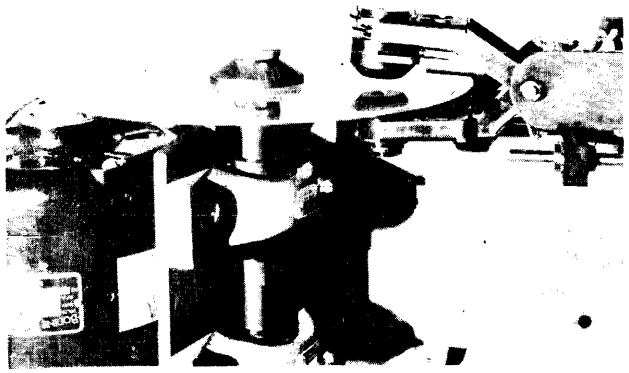


Figure 10—Flexible disc lens mount and air clamp

REFERENCES

1 C H BECKER
UNICON computer mass memory system
AFIPS Conference Proceedings Spartan Books Washington D C vol 29 1966

2 R L LAMBERTS G C HIGGINS
A system of recording digital data on photographic film using superimposed grating patterns ibid
ibid

3 J D KUEHLER H R KERBY
A photo-digital mass storage system ibid

4 J L CRAFT E H GOLDMAN W B STROHM
A table look-up machine for processing of natural language
IBM Journal of Research and Development 5 3 1961

5 W DRUMM
Photo optical recording of digital information
presented at SPSE Symposium on Photography in Information Storage and Retrieval Systems Washington D C 1965
Itek Corporation report 1965

6 R L LIBBY R S MARCUS L B STALLARD
A computer system for non-numerical information processing
Technical Report Itek Corporation February 1967

7 A K CHITAYAT
Means and Methods for Marking Films
U S Patent Number 3 266 393 August 16 1966

8 C O CARLSON E STONE H L BERNSTEIN W K TOMITA W C MYERS
Helium-neon laser: thermal high-resolution recording
SCIENCE vol 154 3756 p 1550 23 December 1966

9 F FORLANI N MINNAJA
A proposal for a magneto-optical variable memory
Proceedings of the IEEE 54 4 p 711 1966

10 G W KING G W BROWN L N RIDENOUR
Photographic techniques for information storage
Proceedings of the IEEE 54 4 p 711 1966

A rugged militarily fieldable mass store (MEM–BRAIN file)

by W. A. FARRAND, R. B. HORSFALL,
N. E. MARCUM and W. D. WILLIAMS
Autonetics, A Division of North American Aviation, Inc.
Anaheim, California

INTRODUCTION

Military systems* often operate on a data base larger than is economically feasible for internal main store. Therefore, they require the use of auxiliary storage. There are many forms of auxiliary store. Each has characteristics which make it the optimum choice for certain tasks. Factors characterizing the domains of utility of some auxiliary storage systems are:

Fixed Head Drums	Under 4 million bits
Magnetic Real Tape Systems	Over 1/2 sec. average random access/ block
Woven Wire Systems	Under 10 million bits
Tape Loop Systems	Under 50 million bits
Roving Head Drums	Under 50 million bits
Tape Strip Files	Over 1/2 sec. average random access (unit documents)
Disk Pack Systems	20 - 50 million bits/ pack (interchangeable data base)
Fixed Head Disk Files	Under 500 million bits
Roving Head Disk Files	Under 100 ms average access and over 50 million bits

Military data systems studies at Autonetics show a recurring need for a large-scale data file. The required characteristics are: over 1,000,000,000 bits of storage, high data rate, low weight, low volume, long MTBF, and simple maintenance under military field conditions. Our survey uncovered no equipment with these properties, or equipment whose inherent design characteristics could be sufficiently upgraded. Therefore, Autonetics set out to develop a militarily fieldable auxiliary store with a two gigabit data capacity, a data rate of ⅓ million characters per second, and a maintenance schedule of less than one hour per thousand hours of operation, which would pass through a 24-inch square opening. For this, a roving head disk file was chosen.

System

Major differences in electromechanical form with respect to conventional disk files were necessary in order to meet project requirements. In common with conventional units, this device includes a rotating precision disk stack and moving arms carrying sets of heads for access to the disks. However, the basic principle of design is different in that maximum emphasis has been placed on maximizing stiffness to mass ratios in the mechanical structure and in the use of prestressing for improvement in dynamic response and reduction in weight. These principles have resulted in the adoption of an inside-out design incorporating a new disk form consisting of a thin annular foil in tension. Kinematic principles of design have been stressed in the development of this file. Because of the disk form, the new unit has been christened the "MEM-BRAIN File."

A number of fixed head tracks in various configurations are included as an integral part of the disk file. Fixed head read/write techniques are used for various processing operations and for interfacing with diverse processing and peripheral units. Operations provided include automatic and continuous on-line file maintenance, record manipulation, formatting, editing, sorting, merging and collating. Some tracks are used with single read heads for timing and format control. Other tracks provide short recirculating registers for temporary storage of record/request data and format control bands, precession loops for use as push-down reg-

*Examples: Message Switching Centers, Tactical Data Systems, Command and Control Systems, Combat Information Centers, Military Supply Control Centers, Coastal Defense System Data Nets, Air Traffic Control Centers, and other large-scale electronic Data Processing Systems (both military and civilian).

isters, and multihead tracks for internal buffering of data.

System optimization studies showed that greater through-put could be accomplished if certain bookkeeping and formatting features were included in the file proper. By placing these under storage program control, the master control program is freed for more complex functions. Means have also been provided to permit selective positioning of up to four sets of heads simultaneously, while continuing to read or write on other sets. Direct communication between MEM-BRAIN and other peripheral devices may be accomplished without involving the central processor.

Very careful attention to system engineering techniques was needed to optimize the configuration of the MEM-BRAIN File to meet the desired objectives, because of the intense interaction of various design elements. For example, surface density of record cells depends on lineal density along track, radial track density, and zoning geometry. To avoid timing problems resulting from mechanical deflections under service conditions, low lineal density was desirable. The maximum lineal track density (500 bits per inch) was chosen to allow inexpensive exploitation of precession loops and their inherent data manipulation efficiency. Such operation would not be possible if self-clocking data recording were used. Military vibration environments would require self-clocking to permit use of higher lineal cell densities. Development of a track-seeking servo allowed the necessary compensating increase in track density, both by eliminating need for intertrack separation, and by allowing use of smaller track width. These factors reacted to reduce the size and mass of the moving arm elements (among other things) which in turn modified the criteria for lineal density along the track, to "Close the Loop." Similar interactions of design elements occurred at almost every turn in the development of the MEM-BRAIN File. Thus every design idea required checking not only for its direct effect on the immediate problem, but for its influence on every other facet of the entire system.

Mechanical design

The mechanical design of MEM-BRAIN centers about the high stiffness to mass ratio of a pretensioned membrane both for the disks and for the primary support structure of the rotating mass (the disk stack). This principle leads to an inside-out form of the file. The disks are axially stacked, mounted, and rotated within a hollow shaft and accessed from the center, as shown in Figure 1. The disks are thin annular sheets of magnetically plated foil stretched taut on tensioning hoops which are attached to the external tubular shaft. The tensile preload established by this process causes the plated membrane to act as a rigid surface of uniform flatness. This eliminates surface finishing operations. Such a membrane has tolerance to extreme thermal variations and aging without distortion out of plane because the stress level remains positive and less than the elastic limit. The disk has a mass two orders of magnitude less than a conventional disk and has very low mechanical Q. These properties of the disk are basic to the militarized qualities of the MEM-BRAIN File.

The disk stack forms a cylinder which is closed around the periphery. This disk stack is directly driven by a synchronous motor which combines with the high moment of inertia to mass ratio of the disk stack to provide a highly uniform rate of rotation. The enclosed disk periphery reduces air flow and the resultant noise level in the vicinity of the file.

One end of the disk stack is supported by a pretensioned pair of shallow-coned metal sheets forming a light end-bell which is extremely rigid, both axially and radially. This end-bell is carried by a bearing supported by a similar conical pair. This structure is indicated in Figure 1. Three degrees of freedom are thus constrained. The other end of the disk stack is supported by a bearing through a membrane structure to complete a five degree of freedom constraint. The sixth degree of freedom is the desired rotation.

A central, nonrotating rigid tubular member carries the protruding head support arms. These arms include a relatively massive and rigid fixed frame and a light but stiff moving element which carries the read/write heads. The frame and moving element are locked together during reading and writing operations, so that the critical circumferential location of the heads is maintained by the fixed frame. A set of eight heads per arm are mounted in pairs on four gas-lubricated sliders, to access two facing disk surfaces. The sliders are forced toward the disk surface to maintain an essentially constant gas lubricated gap. These heads are radially spaced at a separation corresponding to the radial width of recorded pattern (one band). They are constrained to move with the movable arm element. Thus, the servo need only move through one band to reach all records on a facing pair of disk surfaces.

The large taut-disk-stack, central tube, and clamped arm comprise an extremely stiff, low-mass, low Q, mechanical assembly. This can withstand shock and vibration exceedingly well even while operating. The unique kinematics and self-set head positions insure a high quality information storage and retrieval system in spite of environmental extremes. The total mechanical system described provides a mechanism having high natural frequency and low vibration amplitude. By judicious application of visoelastic coatings, the me-

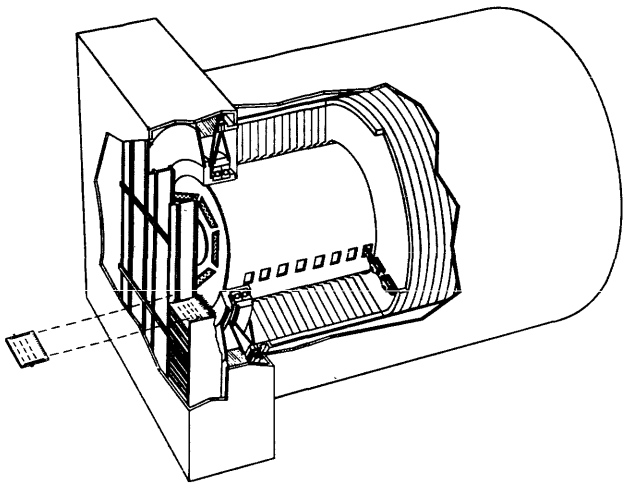


Figure 1—Two gigabit MEM-BRAIN file

chanical Q of the total structure is kept below one.

To provide for control and processing functions, a group of fixed heads are located at one end of the disk stack. These are gas-bearing supported against the exterior surface of the end disk. The head units are supported by rigid structural members attached to the central structure and are located and spaced to provide rapid access and buffer loop capability as required for file maintenance and manipulation. In function, the support for these fixed heads is equivalent to a solid headplate such as has been used in Autonetics' computer memories.

Head arms

Each arm is capable of radially positioning a set of eight heads. The positioning is done by a servo mechanism whose control signal includes track number and track center data derived from the read head signals. Therefore, the servo is guided by true head position to track center instead of arm position. This permits the use of a lighter (under one ounce) head transport than is possible where the position signals are derived at a remote portion of the arm structure. The head servo is optimized by the minimum mass (therefore, minimum inertia) design. Tolerance to temperature and thermal gradients is also improved. To permit servo switching, a passive mechanical clamp is provided to hold each arm accurately at the selected track center position.

The arm is light enough so that it can be driven by a direct drive electric motor at force levels up to 40 G's, with negligible reaction on the central column. This, coupled with the mechanical clamp on each arm, allows reading or writing on any head set without concern for the motion of any other arm. Each arm is, therefore, independently positionable, and a positioning-transcribing interface can be set up with head sets giving a continuous flow of data. The positioning time is never more than one disk revolution (50 milliseconds) and, therefore, no a priori ordering of channel seeks need be used. The arm positioning time reduces to 6 milliseconds for adjacent track positioning. The positioning time includes assignment of a servo electronic set, release of the arm clamp, traverse to the proper track, settling on the track center, verification of track center, clamping of the arm, reverification of track center, and release of servo electronics to other functions.

Electronic design

The basic storage function is accomplished on the disks accessed by the positionable heads. The built-in control functions are implemented by the fixed head elements and associated electronics. The clock and origin are unalterably recorded on fixed head tracks. The fixed head elements with associated electronics are used for implementing the control and processing functions. Occasional coordinated use of the I/O buffers may also occur in executing certain of the internal processing commands.

Except when they are in use for internal processing, the I/O buffers provide for asynchronous data transfer between MEM-BRAIN and other equipment, in such serial or parallel form as may be required by the application.

Internally, all operations are synchronized by the clock. Such operations are basically serial in nature, but many may be carried out serial by character, rather than serial by bit. Normal NRZ recording is used in the fixed head area. A specially modified form of NRZ recording is used in the movable arm (main storage) area.

The electronics include all necessary elements to carry out storage and processing functions short of actual arithmetic operations. Such elements include read and write amplifiers, multiplexing networks, comparison logic, control registers, servo electronics, and buffers.

Four I/O control registers are used to direct internal operations in response to commands as more fully described later. Four sets of servo electronics are also provided to control arm positioning operations. On an "as available" basis, any control register may use any servo unit to control any of the 60 arms, and may make any required logical connections to transfer data between that arm and any specified combination of its I/O data channels. External data transfer normally occurs on a ready-strobe basis.

The servo amplifiers respond to a signal developed by detecting the phase difference between adjacent

tracks. The effectiveness of this technique is improved by a special form of NRZ recording. The arm is slewed for gross movements by noting the tracks traversed. Fine positioning is controlled by adjacent track nulling. A track identification recorded in each sector (eight bits) is used to verify track centering.

All circuitry consists of advanced state-of-the-art replaceable microminiaturized circuit board assemblies designed to withstand military environment with excellent MTBF and MTTR.

I/O control

The internal organization of the MEM-BRAIN File is described in the functional schematic, Figure 2. Four control registers, each associated with its I/O data channel block, simultaneously control the positioning of any head arm and additionally the switching of the heads associated with any arm (8 heads) to the I/O data channels normally via buffers. They also accept and transmit the instructions in control of internal data manipulation.

The controller acts upon discrete control lines to commit instructions and data and to deliver MEM-BRAIN status as required. A Security Code provision is available as a part of the instruction word. When used, this code requires a match to allow the function to proceed.

Four I/O buffers each present digital inputs and outputs to the duplex interface data channels. These channels are capable of asynchronous repetition rates considerably in excess of most CPU's. Therefore, instructions and data may be transferred at the maximum rate of the CPU within the capacity of the static buffer. The only other limitations on rate are the MEM-BRAIN (337 KC) clock, and processing time if MEM-BRAIN processing is involved.

Special factors such as busy conditions of head arms and priorities by other connected CPU's, are determined internally and expressed indirectly through the output control lines. The internally stored program of the MEM-BRAIN provides for most adjustable control functions. Differences in the discrete control line requirements of different CPU's need only the replacement of a card to provide compatibility of the MEM-BRAIN File interface.

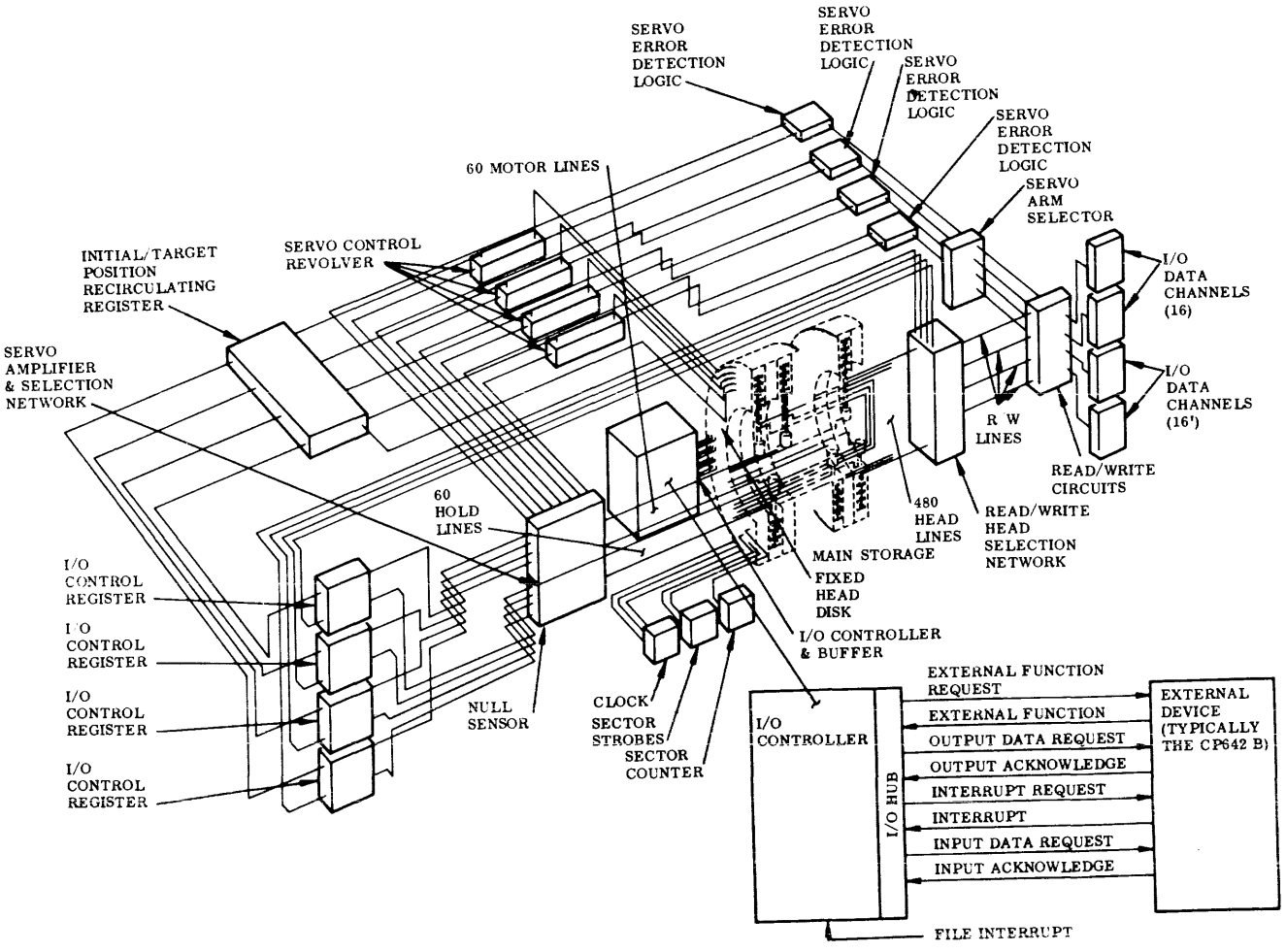


Figure 2—Functional schematic of 60-arm digital disk file

Design for reliability

Design principles embodied in the MEM-BRAIN File have been selected with the ultimate reliability of the device in a military environment as a paramount consideration. Simplicity is always of extreme value in eliminating failure modes and enhancing reliability of any system. Consequently, at every point where a choice between simple and complex solutions was available, the simpler technique was chosen.

For any unit intended for military application, it is desirable to assume the need for operation in a hostile environment. Consequently, ruggedness is a necessary feature in such a design. In the MEM-BRAIN development such ruggedness has been sought at every decision point, but not by the brute force method of increasing structural dimensions. Rather, the technique has been one of reducing mass wherever possible and increasing effective stiffness and structural damping to minimize response to shock and vibration.

The electromechanical elements of such a device are known to be sensitive to adverse environments related to humidity, pressure, and temperature. In the MEM-BRAIN File, therefore, the electromechanical package has been designed as a hermetically sealed unit, and care has been taken in the choice of materials and their structural interconnection to minimize distubrance of critical locations due to temperature or temperature gradient.

The electronic elements of such a system, on the other hand, may be more subject to deterioration in performance by parameter drift or catastrophic failure. To minimize the failure potential in the electronic system, the background developed on the Minuteman standard parts is used. Also, the electronic package has been designed for ready access to permit maintenance by module replacement. Its enclosure is dust-proof and drip-proof, but, not hermetically sealed.

SUMMARY

The objectives of the MEM-BRAIN File development as it has been described were threefold: First, the fundamental target of military fieldability; second, the desire to provide exceptional operational capacity and capability; third, to accomplish these in a light compact unit adaptable to mobile and airborne applications. These objectives were all met by consistent application of system engineering methods by team effort. Table 1 summarizes the characteristics attained.

Table 1. Tabular summary of MEM-BRAIN characteristics

A. *Highlights*

Militarized design
High data capacity
Small physical size
Low weight
Hermetically sealed magnetic recording
One complete package
Interface to any computer
Internal stored program control
Internal buffering
Easily programmed
Short access time
High data transfer rate
Multiple independent head access
Transcription permitted while a different arm moves
Microelectronics
High MTBF
Excellent maintainability

B. *Detail Characteristics*

Capacity (bits)	15×2^{27} (approximately 2,000,000,000)
Size (inches)	$23\times23\times41$
Weight (lb)	400
Power (watts)	1500
Head transports	60 independent
Maximum head transport time	50 milliseconds
Adjacent track head transport time	6 milliseconds
Disk rotation time—nominal	50 milliseconds
Data cells per track	16,384
Tracks per band	256
Bands per surface	4
Heads per band	1
Surfaces per head transport	2
Simultaneous head transports	4
Simultaneous I/O channels	4
Maximum cell density	500 bits/inch
Radial track density	200 tracks/inch
I/O channel rate—nominal	327K bytes/seconds
Environment	MIL-E-5400, MIL-E-16400 MIL-E-4158, FED-STD-222

North American Aviation, Inc., provided support in the development of the MEM-BRAIN concept and the fabrication of a 30 megabit test and demonstration model. This model has demonstrated the basic principles and design details needed for construction of the full scale File. Complete design and fabrication of a 2 gigabit unit is under way.

The "Cluster"–four tape stations in a single package

by JOHN T. GARDINER
Burroughs Corporation
Pasadena, California

INTRODUCTION

The familiar shape of the magnetic tape transport has undergone a dramatic change. As a result, space age television shows may have to find convincing replacements for madly spinning banks of vertically placed reels. The imagination of the public may suffer, but the industry is provided with a new peripheral having a most exciting future.

Details of a multistation Magnetic Tape Cluster are presented in this paper to provide an insight into how it was conceived, how it is built, and its capabilities.

The Cluster concept

If we reflect for a moment to consider the scheme generally used in digital tape drive design, one common characteristic becomes apparent; the tape path formed by the reels, the transducer and the tape buffers lies in the same plane. This is the simplest and most direct way to accomplish the objectives of a single station machine. Of all the transports on the market today, perhaps 80% are designed around the formal vertical plane tape path as shown in Figure 1a.

Another popular method places the tape path in a horizontal plane such as shown in Figure 1b. This design is advantageous in many respects and offers convenient table height operation. Some forward thinking designers have combined the two ideas to provide two transports in the same box (Figure 1c).

Our earliest considerations for designing the multistation Cluster Unit contained several features which have remained throughout the entire development. These requisites included a four station package which shared electronic and mechanical features to achieve the best possible combination of performance, reliability and low manufacturing cost.

Naturally, the development group first took a hard look at the commonly used vertical plane geometry. It was found that some improvement was possible by rearranging reel positions similar to those shown in Figure 2a. This arrangement could share electronics very nicely but the ability to share mechanical components was questionable. Figure 2b shows a folded version of the above arrangement which improved the possibility of sharing mechanical features.

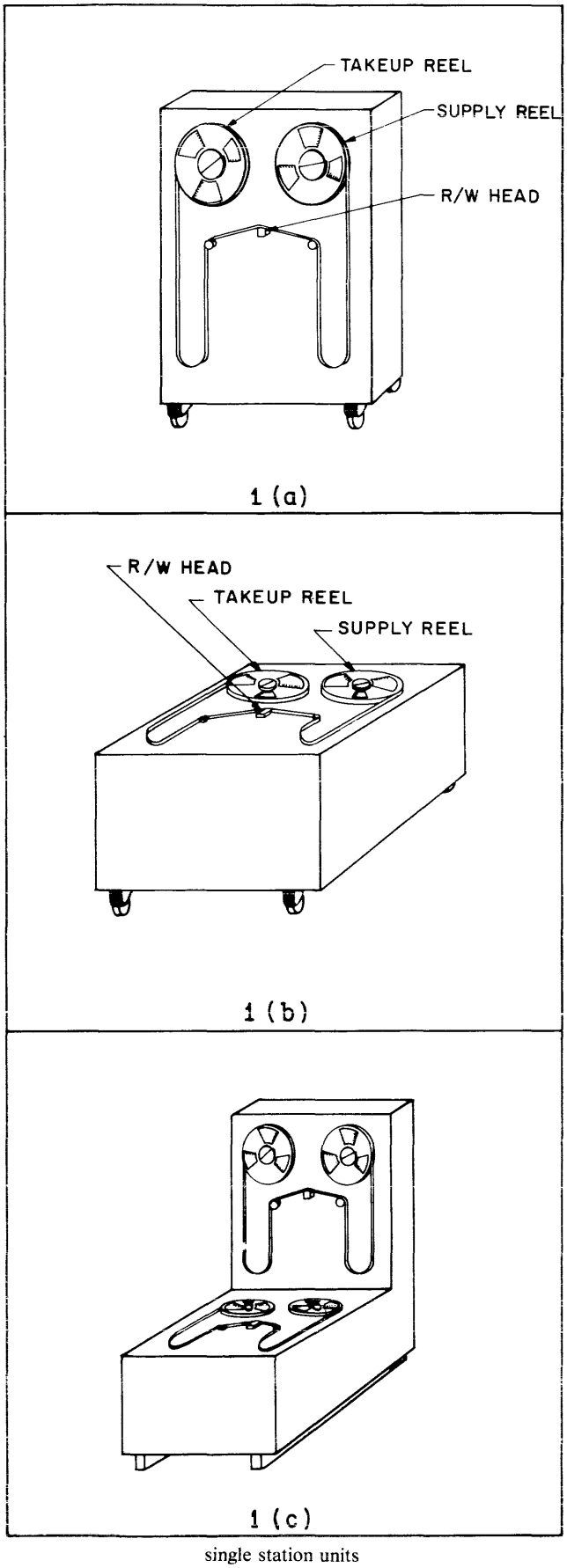
An original design, Figure 2c, having a multiplicity of vertical tape paths but with horizontal tape buffers, received attention during the early development period. It was apparent, however, that a "sandwich" machine such as this would be plagued with severe accessibility problems.

For each of the above designs there were advantages which could be cited, but one fact remained abundantly clear: *none* of the arrangements represented the major step forward we were seeking. We realized that the point had been reached where additional time spent in simply rearranging tape path components was not justified, and a clear departure from established custom was a "must" if our objectives were to be met.

What then could be done to break this impasse? We were confident that the reels themselves were the key factor and, since a simple rearrangement did not fulfill our needs, the logical remaining possibility was to stack the reels and operate them on a common axis. This possibility proved to be the turning point and has since become the foundation of the multistation Cluster design.

Stacked reels, in themselves, are certainly not original with this design. They have been used in the field of music reproduction and, to some extent, in the digital field at very low or incremental tape speeds. What is original with us is the design associated with making the philosophy work on a multistation tape transport operating at relatively high tape speeds and recording densities.

Once the general approach had been determined, the next step was to examine tape path configurations capable of transporting tape which terminated at two levels of spooling. Refinements came rapidly,



single station units

Figure 2—Conventional tape path geometry for coplanar multiple station units

and one particular path emerged which seemed to have superior characteristics over all others.

A series of preliminary layouts was made and, much to our delight, the various pieces began to fit together with remarkable solidarity. Within a month we felt that sufficient understanding had been obtained to proceed with a full-fledged development program, and management gave its unqualified approval.

Thus, the multistation coaxial reel concept was born. Three months later, a hurriedly constructed experimental model, Figure 3, was providing operational data. It is gratifying to note that the tape path shown here has stood the test of time and basically is the same as that used in the machine as it exists today. The current design is shown by Figure 4. It is a four-station Cluster which operates at a tape speed of 45 inches/sec. The Cluster is now in production and will be available to industry within the next several months.

Geometric relationships

Before proceeding with a discussion of the finished product, let us examine some of the geometric relationships associated with nonplanar tape paths.

We begin by considering two coplanar reels as shown by Figure 5A. This condition requires that the reels be in a spaced-apart relationship and that the axes of rotation be parallel. Desired convolutions of the tape between the two reels are a function of judiciously placed pivot points which are installed normal to a plane of reference.

The plane of reference, for our needs, may be defined as a plane which passes through one edge of the tape and is normal to the tape surface. It is particularly desirable to consider the tape as lying in the plane of reference between the point of entry into the supply reel buffer and the point of emergence from the take-up reel buffer. For purposes of our discussion, this condition will be assumed for all cases.

We next introduce a modified condition which retains the parallel relationship of the reel axes but repositions the reels such that they are no longer coplanar. This condition is illustrated by Figure 5B. As can be seen, the pivot points now assume an angular relationship to the reel axes. Convolutions of the tape within the plane of reference (between pivot points A and B) are removed from the scope of our discussion by definition.

We continue our space relationships by repositioning one reel such that its axis becomes an extension of the axis of the second reel as shown in Figure 5C. This move is for convenience only and no significant geometric changes are introduced when compared with Figure 5B. However, a comparison of Figures 5A and 5C leads to several important observations.

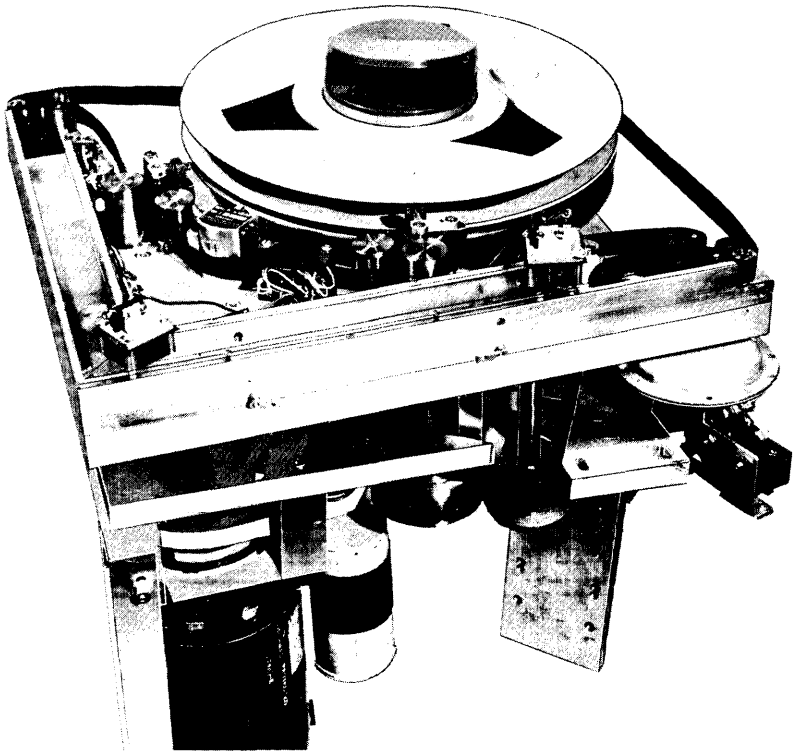


Figure 3 – Coaxial reel breadboard

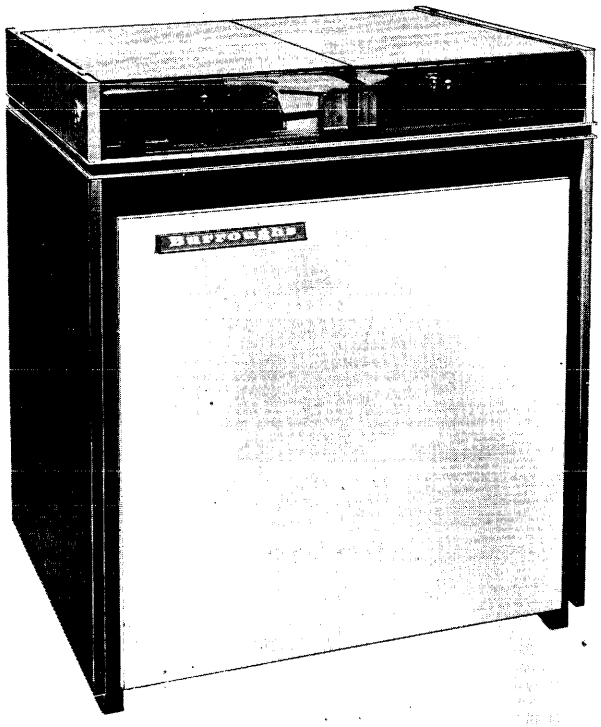


Figure 4 – Burroughs' four station Cluster

1. If the pivot points are cylindrical and fixed, they will perform their intended function under either condition. The relationship between the tape and a fixed pivot is one of sliding contact between the two surfaces but a variation in the contact length of the tape as it proceeds around the pivot will have no ill effects in guiding the tape.

2. If the pivot points are cylindrical and rotatable, they will perform properly under condition 5A only. Under condition 5C, we observe that a variation in the contact length of the tape as it proceeds around the roller must introduce relative motion between the two surfaces. The roller will therefore migrate along its axis or, if restrained, an undesirable slippage is introduced between the tape and roller.

To summarize our position at this stage, we find that rotatable pivots cannot be used in conjunction with coaxial reels unless some type of relationship is established which prevents an angular contact between the direction of tape travel and the rotation of the roller. Since rotatable pivots are highly desirable for maintaining a low frictional drag, and perhaps even more important, to keep the drag forces constant, it is imperative that such a relationship be found.

To accomplish this, we introduce an angularity between the axis of reel rotation and a plane of reference as illustrated by Figure 6A.

Now, consider the tape as it proceeds from the supply reel to rotatable pivot point A. Upon contact, the tape proceeds around the roller without sliding contact. If 180° were selected as the arc of contact, the tape would simple be returned to the reel. Under this condition the bottom edge of the tape would make a "point" contact with the plane of reference.

Suppose we wish the tape to depart from the roller in such a fashion that its lower edge were to continuously lie in the plane of reference. This is accomplished with the aid of an additional rotatable pivot point C, which must be precisely located. The location must be such that a line drawn between pivots A and C must lie on the intersection of the plane passing through the bottom edge of the spooled tape and the plane of reference.

Since by definition we must position the tape surface normal to the plane of reference, we install pivot point C such that its axis is normal to the plane of reference rather than simply parallel to the axis of pivot point A. Under these conditions, a small twist is introduced in the tape over the distance "L". Note, however, that the rule stating that no relative motion is permissible as the tape travels around a rotatable pivot point has not been violated.

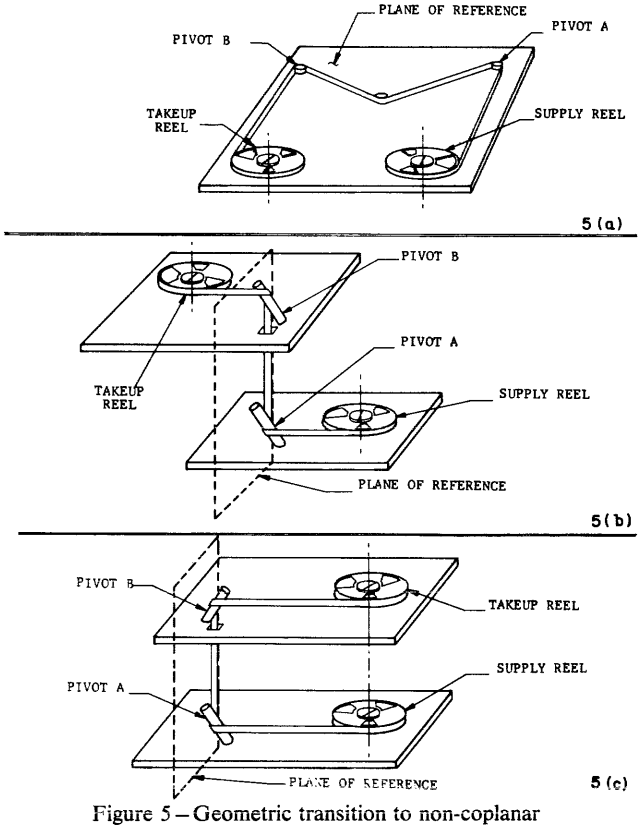


Figure 5 – Geometric transition to non-coplanar tape path – fixed pivots

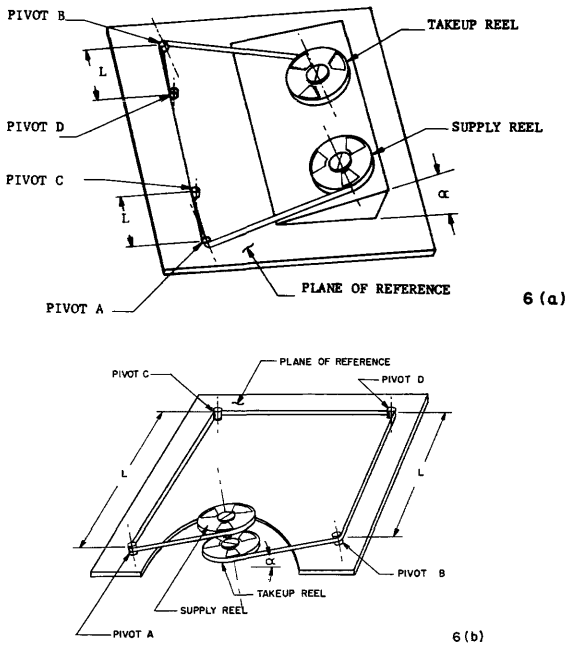


Figure 6 – Geometric relationships of non-coplanar tape path and rotatable pivots

All that now remains to accomplish our geometric objectives for coaxial reels and rotatable pivot points is to reposition the two reels in a stacked attitude, such as illustrated by Figure 6B. We know from the foregoing discussion that this can be done without introducing any significant change to previously established relationships.

We find that for small values of (a), a reasonably short distance "L" can be tolerated without difficulty. The magnitude of (α) is a function of the spacing between reels and the distance from the reel axis to the location of pivot point A (or B). These relationships, as expressed in actual Cluster values, result in a stress of approximately 40 psi at the outermost fibre of the tape. This figure is very low when compared with the proportional limit of the mylar material and a more meaningful comparison may be made in terms of the strain profile introduced across the width of the tape. Tension forces acting on the tape will desirably approximate 9 ounces and the 40 psi fibre stress is found to be equivalent to 0.3 oz. of tension. This results in a profile which has a variance of about 3% across the tape width.

Geometric implementation

At this point it would be well to examine the hardware and see how the geometrics have been reduced to practice. Figure 7 shows the structural relationship between the frame and an unusual looking weldment called the spider. The spider is supported at four points by vibration isolators to damp spurious frame or floor oscillations which might be detrimental to reliable operation. The pads on each arm of the spider provide bearing surfaces for the coaxial reel drive assembly.

One of the four coaxial reel drives is shown in mounted position on the spider. The deck assembly is also shown in mounted position attached to the face of the tubular hub of the spider.

A close look will reveal the angularity (α) which exists between the reel drive and the vacuum columns (plane of reference). This angular relationship is obtained by machining the mounting pads of the reel drive housing at precisely the proper angle. A vertical adjustment is provided in the reel drive to correctly align the height of the reels with the plane of reference.

Figure 8 shows the top of the four station Cluster without covers. This provides a view of the various component parts which lie in the plane of reference. For purposes of clarity, Figure 9 has been prepared to enable us to examine the elements which comprise a single station. Beginning with the supply reel, we trace the tape path as follows:

1. Tape from the supply reel passes around rotatable pivot A (remembering the axis of pivot A is parallel to the reel axis).
2. Between pivot A and rotatable pivot C the tape undergoes a slight twist of about 3°.
3. Since the axes of pivots C and D are normal to the plane of reference, the tape lies in the same

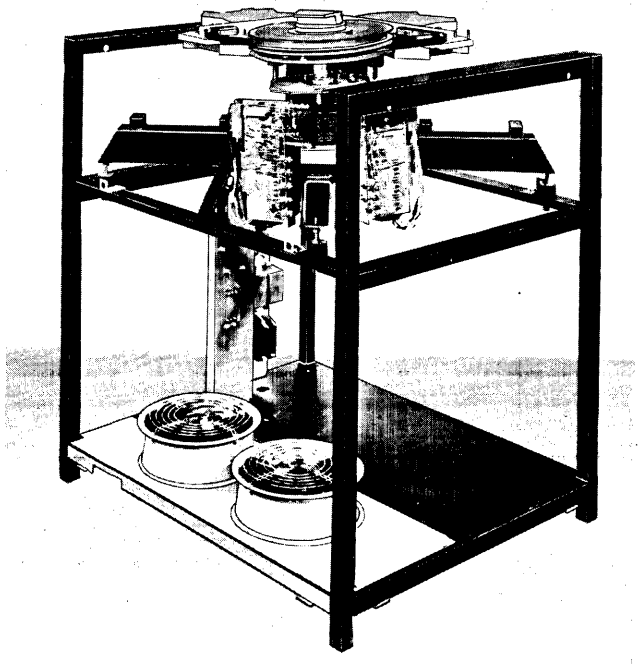


Figure 7 – Hardware implementation

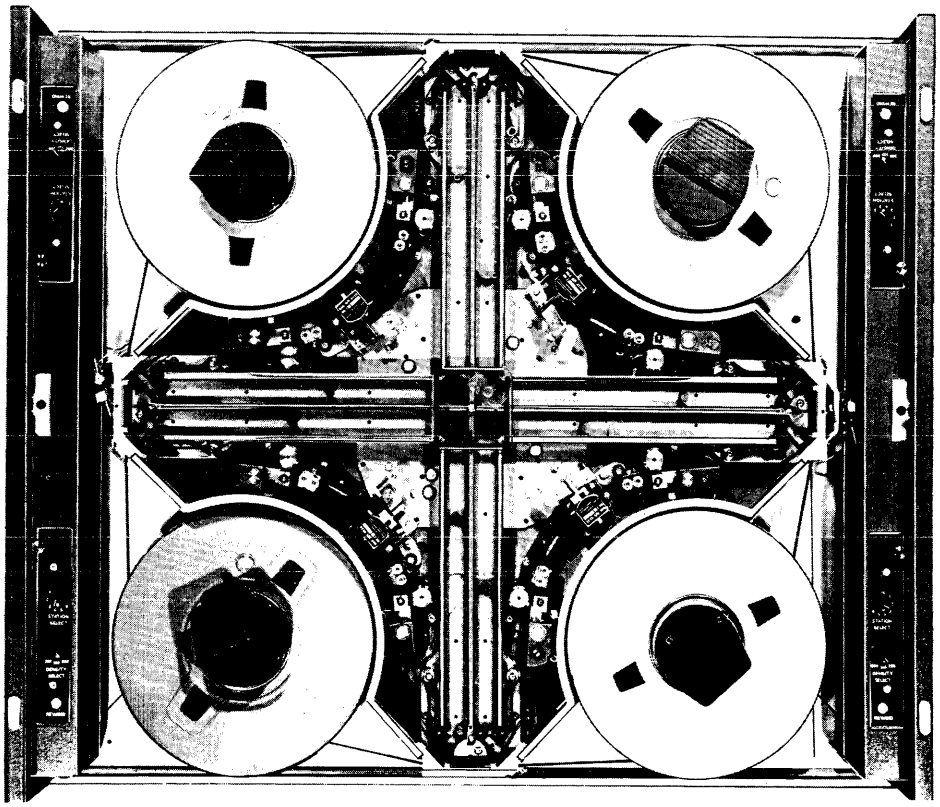


Figure 8—Deck assembly showing quad configuration

plane at every point in the path between these two extremes.

4. The tape now proceeds in a fairly conventional manner from the supply buffer, past the reverse pinch roller driver, and over the left hand cleaner guide. Here, the reference edge of the tape is precisely aligned for passage over the head and any loose oxide particles are scraped off and vacuumed away.
5. Prior to reaching the read/write head, the tape passes through the end of tape/beginning of tape sensor. This sensor is also used to fix the exact position of the tape leader latch when the tape is to be unloaded.
6. As the tape passes over the read/write head, it is held in close contact by a magnetic shield supported by the lift-off arm. The shield not only assures an optimum flux path but effectively damps any vibrations which might be introduced during startup. The lift-off arm rotates to lift the tape completely off the head during rewind, thereby preventing unnecessary wear.
7. The tape continues past the right hand cleaner guide, past the forward pinch roller driver, and into the take-up buffer. A slight twist of the tape is again introduced between rotatable pivot points D and B. At pivot B, the tape is realigned in the plane of the tape reel (remembering that the axis of pivot B is parallel to the reel axis) and finally is spooled onto the take-up reel.

Cluster design features

Speed, size and weight

Each station of the Cluster operates independently at a tape speed of 45 inches per second (forward or reverse). The four stations are packaged in a single box 43″ high, 36″ wide and 30″ deep. This compactness is further exemplified by the weight of the unit which approximates 800 pounds.

Tape is moved by means of sealed pinch roller drive modules which are extremely positive and quiet in operation. Much of the clackety-clack is gone and, once the internal adjustments have been correctly made, further adjustment becomes essentially non-existent. This condition arises from the fact that metal-to-metal contact found in most electromagnetic clapper devices has been eliminated by means of a cushioning device of a proprietary nature.

The capstans for all four stations operate continuously at a synchronous speed of 900 RPM. Power is supplied by a single motor, and heavy flywheels provide sufficient inertia to maintain tape fluctuations at a minimum regardless of pinch roller activity.

Tape guides are located on either side of the head in the conventional manner. The design of the tape

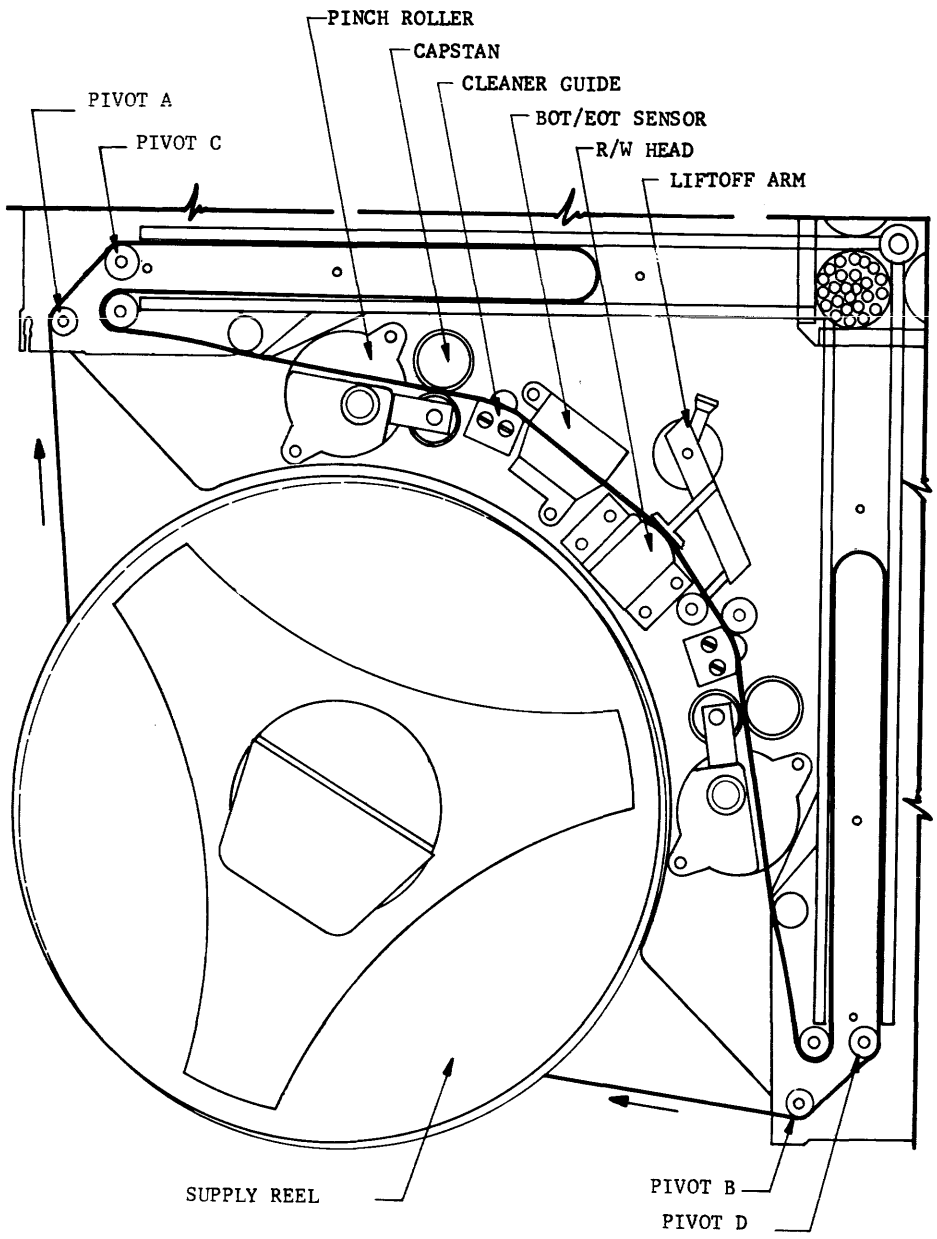


Figure 9 – Tape path and associated components

guides serves a dual function since the guides also provide for efficient cleaning action. Particles removed by the cleaning action are not allowed to accumulate but are swept away in a high velocity air stream.

The vacuum columns are assembled in the form of a cross and receive vacuum pressure from a single source. Vacuum pressure to each station is isolated by means of a fail-safe check valve which closes should tape failure occur. Other stations are thus protected against loss of vacuum pressure and remain operational.

Of particular significance is the fact that the entire tape chamber is a sealed compartment which maintains this critical area dust free. Air within the chamber is recycled 30 times per minute and passes through a micronic filter element on each cycle. The air temperature within the chamber is continuously sensed and compared with room ambient temperature by means of a differential thermostat. A small cooling unit maintains the temperature of the recycled air within ± 1° C of the ambient temperature at all times.

Cluster logic is a hybrid combination of discrete components and integrated circuit chips. Normal input power is approximately 2 KW at 208/230 V, single phase, 50/60 cycle.

The read/write electronics accept either seven or nine channels of information in any code combination.

The external control (Central System) is switched to any one of the four stations by an appropriate exchange within the Cluster. Reading or writing at bit densities of 200, 556 and 800 bits per inch (NRZ1) is provided as standard. At 800 bpi, the transfer rate is 36KB. This capability can be extended to 72KB by installing the Phase Encoding option which is presently being offered at a density of 1600 bpi.

Terminology for a machine composed of one set of read/write electronics, four active stations and nine channel R/W heads is 1×4×9. Under these conditions, any one of the four stations can be independently reading or writing at any given period of time under the direction of a single external control.

Provision is made to install a second set of read/write electronics if desired. In this case, the above terminology becomes 2×4×9. Under these conditions, any two of the four stations can be simultaneously and independently operating under the direction of two external controls.

If a user plans to use the Cluster with Burroughs' B2000 systems, the ability to fully utilize its multiprocessing capability is a highly desirable feature. Since multi-processing will generally require more than four tape stations, the Cluster can also be obtained as a slave unit and will be equipped with two, three or four stations to exactly respond to a user's needs. The slave unit is identical in design to the master unit except for unneeded electronics.

The terminology 2×8×9 is used to define a master and slave combination where any two of the eight available stations are under the independent and simultaneous direction of two external controls.

SUMMARY

The multistation coaxial reel concept is believed to be a significant advance in the art of digital tape transport design. In support of this belief are the many new features of the 45 ips Cluster which have been discussed in this paper. However, todays technology is a hard task master and technical advances without appropriate economic advances are likely to be lost in the shuffle. As most of us have learned, it is no longer sufficient for the engineer to develop "something better"; that "something better" must also cost less.

Economic studies of the Cluster's design are most encouraging in this respect and major cost reductions have accrued by sharing certain portions of the machine's makeup. This condition is also the result of a program where assigned target costs have accompanied each design element almost from its inception.

Consider the two machines shown in Figure 10. The upright unit is typical of many single station tape transports on the market today.

Externally, we note that a four to one savings is directly applicable to the frame, skins, trim and covers. Internally, the ventilation and vacuum subsystems represent the same four to one savings potential. The power supply and capstan drive elements can be considered as a three to one savings. Perhaps the most outstanding single factor of all is the electronics themselves, here a savings of three to one can be conservatively applied.

What does this all add up to? Although a direct cost comparison between the two machines is not completely fair, we do know that the Cluster not only represents something better, but that it also satisfies the corollary that it must cost less.

We look forward to continued refinement of the Cluster concept and foresee its advantages extended to higher speed machines operating reliably at even greater packing densities than heretofore considered practical.

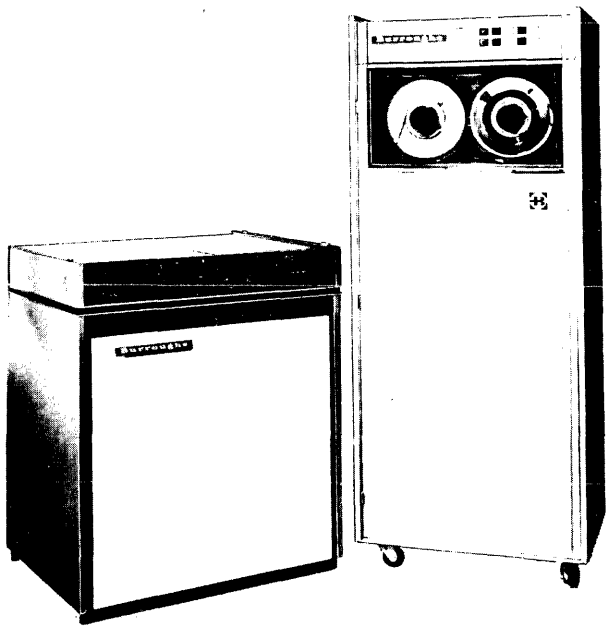


Figure 10 – Multistation Cluster and conventional single station transport

The oscillating vein

by AUGUSTO H. MORENO, *M.D.*
L enox Hill Hospital and New York University School of Medicine
New York, New York

and

LOUIS D. GOLD, *Ph.D.*
Electronic Associates, Incorporated
Princeton, New Jersey

INTRODUCTION

In 1824 D. Barry inserted a glass bulb into the jugular vein of a horse and observed that when the animal was standing the blood flowed intermittently in jets that were not synchronous with either respiratory or heart rates (Brecher[1]). Holt[2] in 1941 arranged a flow circuit where the fluid traversed an interposed segment of collapsible tube and where he could vary at will the ratio of internal pressures. He observed that when the ratio approached zero the tube collapsed and began to pulsate. Brecher[3] in 1952 examined carefully this phenomenon by decreasing the downstream pressure in the vena cava of open chest dogs and concluded that it originated in a momentary complete collapse of a segment of the vein when the extravascular (atmospheric or tissular) pressure exceeded the intravascular pressure. The continuous inflow of blood from the capillaries would then elevate again the intravascular pressure forcing the closed segment of vein to reopen. With repetition of the cycle the vein oscillated at frequencies and amplitudes whose complex relationships he studied in physical analogues made of collapsible rubber tubes. Furthermore, he pointed out the clinical significance of this "chatter" during cardiopulmonary bypass surgery when the collecting reservoir for venous return is positioned at an excessively low level. In 1953 Robard and Saiki[4] and in 1955 Robard[5] reported detailed studies on flow through collapsible tubes and on the clinical implications of its instabilities in relation to apparently anomalous flow-pressure patterns in various vessels of the body. Our own studies[6] on the effect of quiet respiration in a two-chamber-hepatic valve model of venous return indicate that in the intact (closed chest) dog and man the venae cavae operate at the limits of equilibrium and that instability may develop during vigorous respirations.

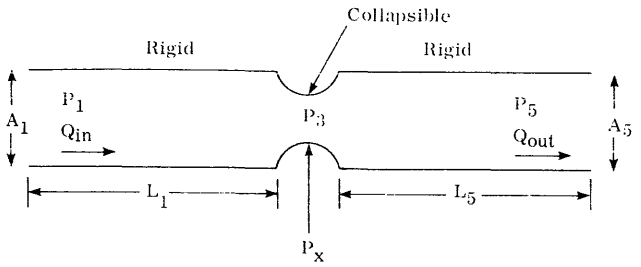
It is of interest that all these authors recognized the significance of the special dynamics of flow through collapsible tubes in the body but complained, at the same time, that the subject has been almost neglected by engineers and physicists. Thus Lambert's model,[7] apparently developed in response to such a need, seems to be one of the first attempts at the mathematical description of flutter in distensible tubes when fluid is driven by steady pressures.

The present report deals with a mathematical formulation and an analog computer simulation of the oscillating vein phenomena. In this paper a dynamic mathematical model is presented and Lambert's predictions[8] of the flutter characteristics are corroborated by the analog computer simulation.

Mathematical description

Very flexible tubes carrying a steady flow of incompressible fluid may be caused to flutter or oscillate when subject to specific steady external pressures. We observe this effect when we impose an external pressure of sufficient magnitude on a section of flexible tubing. When this external pressure is greater than the steady state hydrostatic pressure, an acceleration of fluid leaving this section of tubing occurs. Moreover, the increased external pressure reduces the flow into this section of tubing because of the decreased driving force or pressure gradient for flow into the tube. The net result of these two phenomena is to sharply reduce the volume of fluid contained within this section of tubing. In decreasing the fluid volume within this flexible tubing, the tube diameter also decreases, which in turn acts as a valve, shutting off further downstream flow. The section of tubing above the constriction begins to fill again, due to the upstream pressure. When the hydrostatic pressure at the constriction is equal to the external pressure the tube opens, with a rush of flow downstream, causing an-

other cycle in the oscillation. The idealized situation is pictured below:



The pertinent mathematical equations which describe the system are:

$$\frac{dQ_{in}}{dt} = \frac{k_1 A_1}{\rho L_1}[P_1 - P_4] - \frac{R_1}{\rho A_1} Q_{in} \quad (1)$$

$$\frac{dQ_{out}}{dt} = \frac{k_1 A_5}{\rho L_5}[P_4 - P_5] - \frac{R_5}{\rho A_5} Q_{out} \quad (2)$$

$$\frac{dV}{dt} = k_2 [Q_{in} - Q_{out}] \quad (3)$$

where V is volume in the collapsible section of tubing, Q is volumetric flow rate, P is pressure, and R is the flow resistance.

Moreover, at steady state conditions, $Q_{in} = Q_{out} = Q_{ss}$, and

$$P_{4,ss} = P_1 \left[1 - \frac{1}{1 + \frac{R_5 L_5 A_1^2}{R_1 L_1 A_5^2}}\right] + \frac{P_5}{1 + \frac{R_5 L_5 A_1^2}{R_1 L_1 A_5^2}} \quad (4)$$

where P_4 is the pressure just downstream from the collapsible section of tubing. We may then formulate the dynamic relationship of P_4 as follows:

$$\begin{aligned} P_4 &= P_{4,ss} \text{ when } P_{4,ss} \geqq P_x \\ P_4 &= P_x \text{ when } P_{4,ss} < P_x \end{aligned} \quad (5)$$

where P_x is the external pressure on the flexible tubing. In addition, we observe from the physical description of the system that the oscillation cycle regains its energy from the constant upstream pressure, which causes a fluid volume buildup and concomitant pressure buildup, until the pressure on the upstream side of the valve equals the external pressure. To include this phenomenom in the mathematical description, we alter the system to include a pressure, P_3, on the upstream side of the valve.* P_3 is a function of the fluid volume in the collapsible section:

$$P_3 = P_{4,ss} \text{ when } P_{4,ss} > P_x; \; P_3 = P_x \text{ when } P_x > P_{4,ss} \quad (6)$$

Therefore, Equation (1) may be modified to:

$$\frac{dQ_{in}}{dt} = \frac{k_1 A_1}{\rho L_1}[P_1 - P_3] - \frac{R_1}{\rho A_1} Q_{in} \quad (7)$$

Moreover, until the fluid flow has built up to a volume V* in the collapsible section of tubing, the pressure just downstream is less than the external pressure, P_x, or

$$\begin{aligned} P_4 &= P_{4,ss} \text{ when } P_{4,ss} > P_x \\ P_4 &= P_x \text{ when } P_{4,ss} < P_x \textit{ and } V \geqq V^* \\ P_4 &= P_5 \text{ when } P_{4,ss} < P_x \text{ and } V < V^*; \end{aligned} \quad (8)$$

Equations (2-8) are a complete description of the physical system under consideration. Figure 1 is the analog computer diagram for the simulation of these equations on an Electronic Associates, Inc. TR-48 analog computer.

DISCUSSION OF RESULTS

According to Lambert's theory,[9] the frequency of the oscillation is proportional to:

$$f = \frac{\sqrt{A_5 (P_x - P_{4,ss})}}{8 L_5} \quad (9)$$

This relationship is questionable since it neglects the frictional resistance term and requires quite a few approximations before the final form, Equation (9), is obtained. However, if we follow Lambert's formulation and include the frictional resistance, we obtain essentially the same conclusions as are implied in Equation (9), namely:

(1) The frequency of oscillation increases as the downstream cross-sectional tube area, A_5, is increased;

and, (2) The frequency of oscillation is decreased as the tube length of the downstream section, L_5, increases.

These predictions were verified when the mathematical system describing the oscillating vein, Equations (1-8), were simulated on an EAI TR-48, using the circuit shown in Figure 1. Figures 2, 3 and 4 present the results of the simulation. From these results, it may be concluded that the mathematical system postulated for this phenomenon gives physically correct responses as they compare favorably with Lambert's theoretical and experimental findings.[10]

SUMMARY AND CONCLUSIONS

This paper demonstrates once again the attractiveness of the analog computer in the investigation of process dynamics. Using the most salient feature of the analog computer, namely simulation by (electrical) analogy, we were able to "mathematize" an extremely complicated system merely by providing an accurate

*That is, when the portion of flexible tubing is collapsed, P_3 is the fluid pressure just upstream from the collapsed portion of tubing.

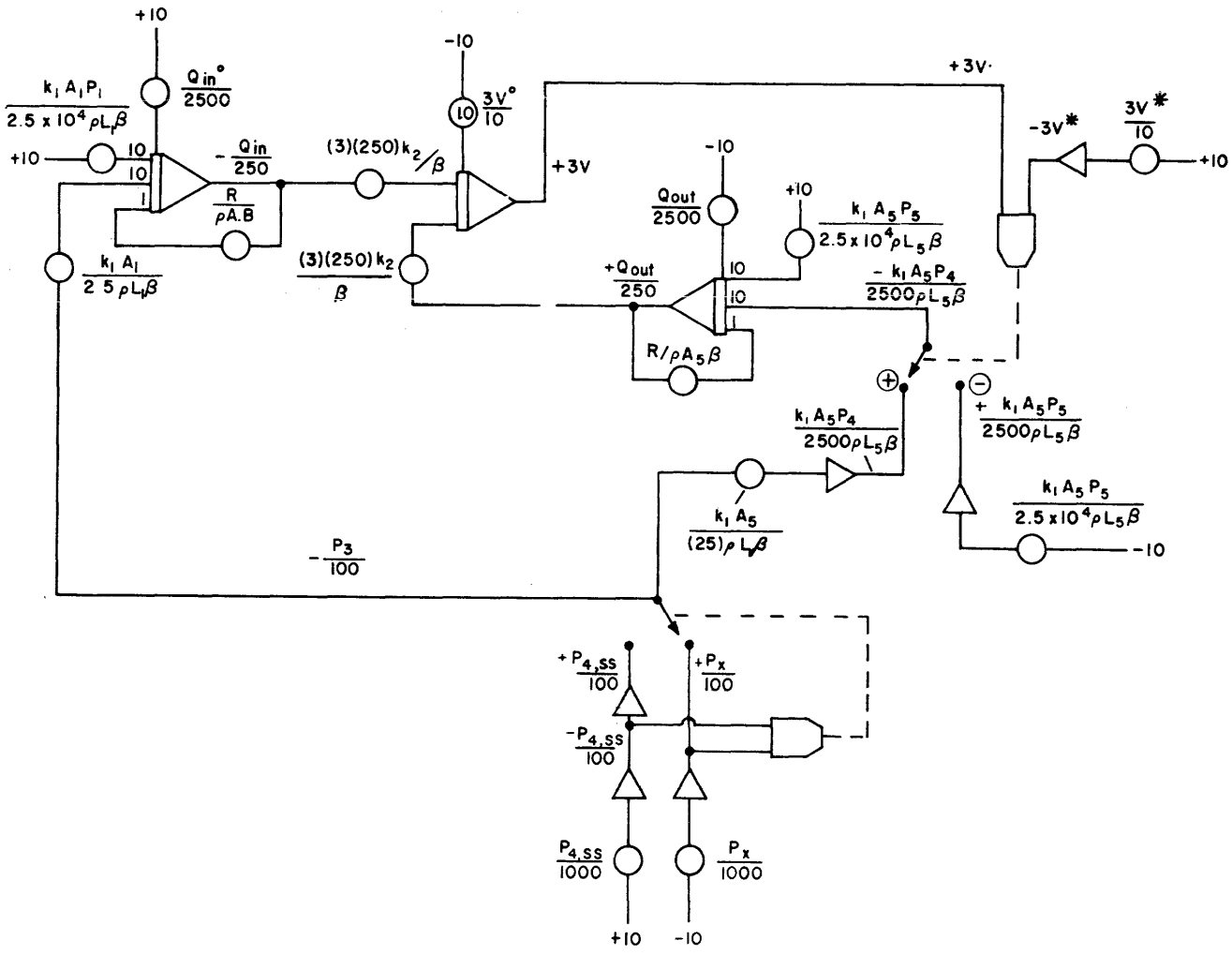


Figure 1—Analog computer circuit diagram

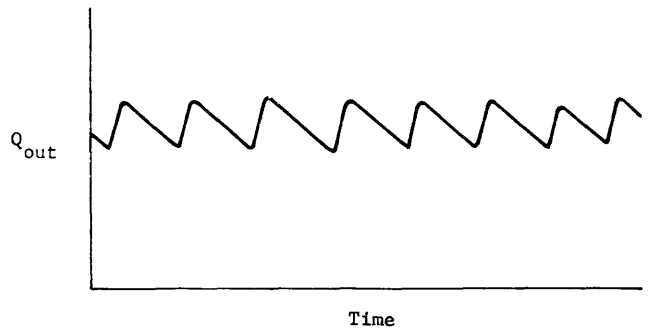


Figure 2—Increasing P_x above $P_{4,ss}$

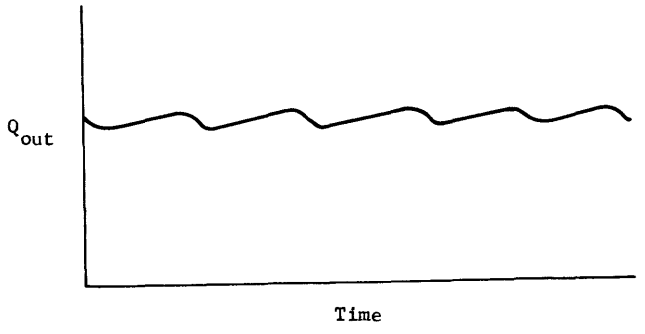


Figure 3—Increasing P_x above $P_{4,ss}$ and increasing the downstream tube length

physical description of the phenomenon under consideration, and then simulating directly this physical description. It is felt that this approach is particularly valuable in the biomedical area where one often deals with physical systems which are not amenable to vigorous mathematical description.

REFERENCES

1 G. A. BRECHER *Venous return* Grune and Stratton, New York Chap. 1, p. 8 1956

2 J. P. HOLT *The collapse factor in the measurement of venous pressure* Am. J. Phys. 134, 292 1941

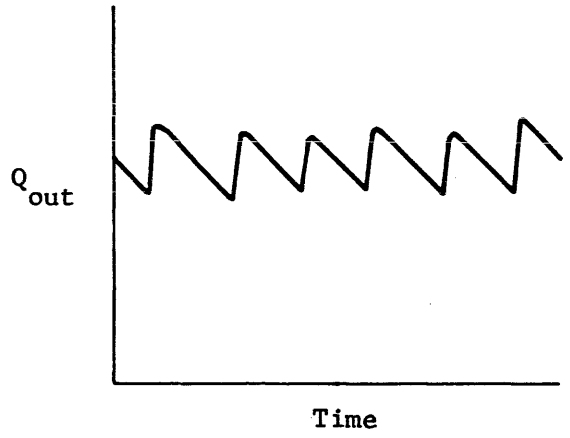


Figure 4—Increasing P_x above $P_{4,ss}$ and increasing the downstream tube cross sectional area

3 G. A. BRECHER *Mechanism of venous flow under different degrees of aspiration* Am. J. Phys. 169, 423 1952

4 S. RODBARD and H. SAIKI *Flow through collapsible tubes* Am. Heart J. 46, 715 1953

5 S. RODBARD *Flow through collapsible tubes: augmented flow produced by resistance at the outlet* Circulation 11, 280 1955

6 A. H. MORENO, A. R. BURCHELL, R. VAN DER WOUDE and J. H. BURKE *Respiratory regulation of splanchnic and systemic venous return* submitted for publication Am. J. Phys.

7 J. W. LAMBERT *Distensible tube flutter from steady driving pressures: elementary theory* paper presented at ASCE, EMD meeting, Washington, D.C., October 13, 1966, Session on Biological Flows

8 Ibid.

9 Ibid.

10 Ibid.

Computer applications in biomedical electronics pattern recognition studies

by A. J. WELCH and J. L. DEVINE
The University of Texas
and
ROBERT G. LOUDON
The University of Texas
Southwestern Medical School
Dallas, Texas

INTRODUCTION

The University of Texas is fortunate in having excellent computational facilities available for utilization in research. This paper is an applications survey dealing with how these facilities are used by the biomedical electronics group in the Electrical Engineering Department of the University.

Two computer systems are involved in the research projects described in this paper. The majority of computation is done on the University's Control Data 6600 system, while a Scientific Data Systems 930 computer maintained by the Electrical Engineering Department provides facilities for data conversion and pre-processing.

The Control Data 6600 system was installed at the University during August, 1966. The center CDC 6601 computer consists of eleven independent, but integrated computer processors. A single control processor which contains ultra high-speed arithmetic and logic functions operates in connection with ten peripheral and control processors.

All eleven processors have access to the central magnetic core memory. This central memory consists of 131,072-60 bit words of storage, each word being capable of storing data 17 decimal digits in length, or two to four central processor instructions. Average instruction execution time in the central computer is less than 330 nanoseconds. Each of the ten peripheral and control processors has an additional 4,096 work core storage. Information exchange is possible between any of the processors and the various peripheral devices.

Peripheral equipment connected to the 6601 at the time of writing of this paper include a CDC 6602 operator console with twin CRT displays, six CDC 607 magnetic tape transports, two CDC 405 card readers (which operate at 1200 cards per minute), two CDC 501 line printers (1000 lines per minute), one CDC card punch (250 cards per minute) and one CDC 6603 disk (75 mega-character storage).

The CDC 6600 is operated by the Computation Center at the University on a closed shop basis while the Scientific Data Systems 930 digital computer maintained by the Electrical Engineering Department is operated on an open shop basis.

The SDS 930 has an 8,192 – 24 bit word core storage memory. Peripheral equipment includes two tape decks which operate at a tape speed of 75 ips with bit density selectable at 200 bpi, 556 bpi and 800 bpi. Additional equipment includes a twelve bit analog-to-digital converter, eight bit digital-to-analog converters and a multiplexer (which in addition to providing coupling for the converters allows input and output of parallel data words and single bit data at logical levels). The A/D converter may be operated at rates in excess of 30 K conversions per second.

For this particular installation, a single input-output channel is available. It may be operated in an interlaced mode (which allows time multiplexing with central processor operation). This input-output channel is independent of the direct parallel input-output capability previously described.

A priority interrupt system is incorporated which allows the computer to operate in a real-time environment. The basic machine cycle time is 1.75 microseconds, with fixed point addition requiring 3.5 microseconds and fixed point multiplication requiring 7.0 microseconds. Single precision floating point addition and multiplication require 77 and 54 microseconds respectively.

Programming languages included in the software package are Fortran II, Real-Time Fortran II, Algol and Symbol (an assembly language).

Real-Time Fortram II was evidently designed to

allow the use of the priority interrupts, but because of poor documentation and inherent weaknesses in the soft-ware, it has been necessary to program most of the conversion routines used in Symbol, either as complete programs or as Fortran II subroutines. A display scope is a part of the system, but the memory location required by the priority interrupt level which it utilizes is also required by the Fortran II system, thereby effectively precluding its use. This handicap has been overcome to some extent by the use of the D/A converters in conjunction with a conventional oscilloscope. This facility, used with a strip chart recorder or X-Y plotter fills the gap left by lack of a plotter or high-speed input-output equipment (input-output is by means of an online teletype or paper tape).

A first step in the data reduction required by the projects described in this paper is the digitizing of recorded analog signals and subsequent storage on digital tape. The SDS 930 provides this capability, but its relatively slow floating point operations and limited memory size make it desirable to use a more powerful installation for actual computation. This need is fulfilled by the CDC 6600 system.

Two projects presently in progress within the biomedical electronics group of the Electrical Engineering Department at the University typify the profitable usage of the computational facilities available. These are a chick dietary deficiency study and a study of audio waveforms resulting from the cough reflex. Both are concerned with time-varying signals and have as a final objective the implementation of a pattern classification algorithm. From a data-handling viewpoint they represent extremes. The chick study works with the electroencephalogram, a relatively slowly varying signal which has a highest frequency component in the vicinity of 150 hz. The cough study operates on a waveform in which the highest significant frequency is in the range of 8 khz. In the first case a data sampling rate of 300 samples per second will suffice to completely represent the waveform, while for the latter a rate in excess of 16000 samples per second will be required. In practice a somewhat higher sampling rate than the Nyquist rate quoted is employed. The quantities of data which must be dealt with differ correspondingly.

Prior to outlining the particular research projects, a brief review of the pattern classification techniques common to both is in order.

Classification theory is divided broadly into statistical (or parametric) decision theory, which treats the data to be classified as being probabilistic in nature, and non-parametric decision theory, which treats the data as being deterministic.

Statistical theory has the advantage that it is possible to obtain a rule for classification that is optimum under stated conditions. These parametric methods require knowledge or good estimates of the underlying density functions and a priori probabilities of occurrence, or as in the case of optimum linear filtering, require statistical stationarity. These properties are rarely known when one works with experimental data, and the underlying joint probability density functions are often so complex that they defy mathematical manipulation unless simplifying assumptions are made.

Most of the non-parametric classification methods cannot be proved to be optimum and must stand by virtue of their success in classifying data from a particular experiment.

In either of the above cases, a preliminary decision must be made as to what parameters are to be measured and used in the classification process. Little theory exists to indicate which parameters should be chosen, and an educated intuitive choice based on the preliminary data from the experiment is probably as good a route as any. In the particular case of biomedical pattern classification studies, the physiological models of the phenomena point toward some features which may logically be expected to contribute to successful pattern classification. Conversely, successful implementation of pattern recognition techniques utilizing measures suggested by the physiological model tend to bolster the validity of such a model.

A pattern classifier has as inputs the measures gleaned from the experiment expressed as an ordered sequency of d real numbers. This sequence is called a "pattern" and may be conveniently thought of as a vector extending from the origin to a point in d dimensional space. The pattern classifier sorts the patterns into categories, for purposes of this paper, R in number. The R point sets constituting categories are separated by "decision surfaces" which may be expressed implicitly as a set of functions containing R members.

Let the "discriminant function" $g_1(X)$, $g_2(X)$,..., $g_R(X)$ be scalar, single valued functions of the pattern X continuous at the boundaries between the R classes. These functions are chosen so that for all X in R_i:

$$g_i(X) > g_j(X) \qquad i, j = 1, 2, \ldots, R \quad i \neq j \qquad (1)$$

i.e., the ith discriminant function has the largest value in region R_i. The surface separating the contiguous regions R_i and R_j is given by:

$$g_i(X) - g_j(X) = 0 \qquad (2)$$

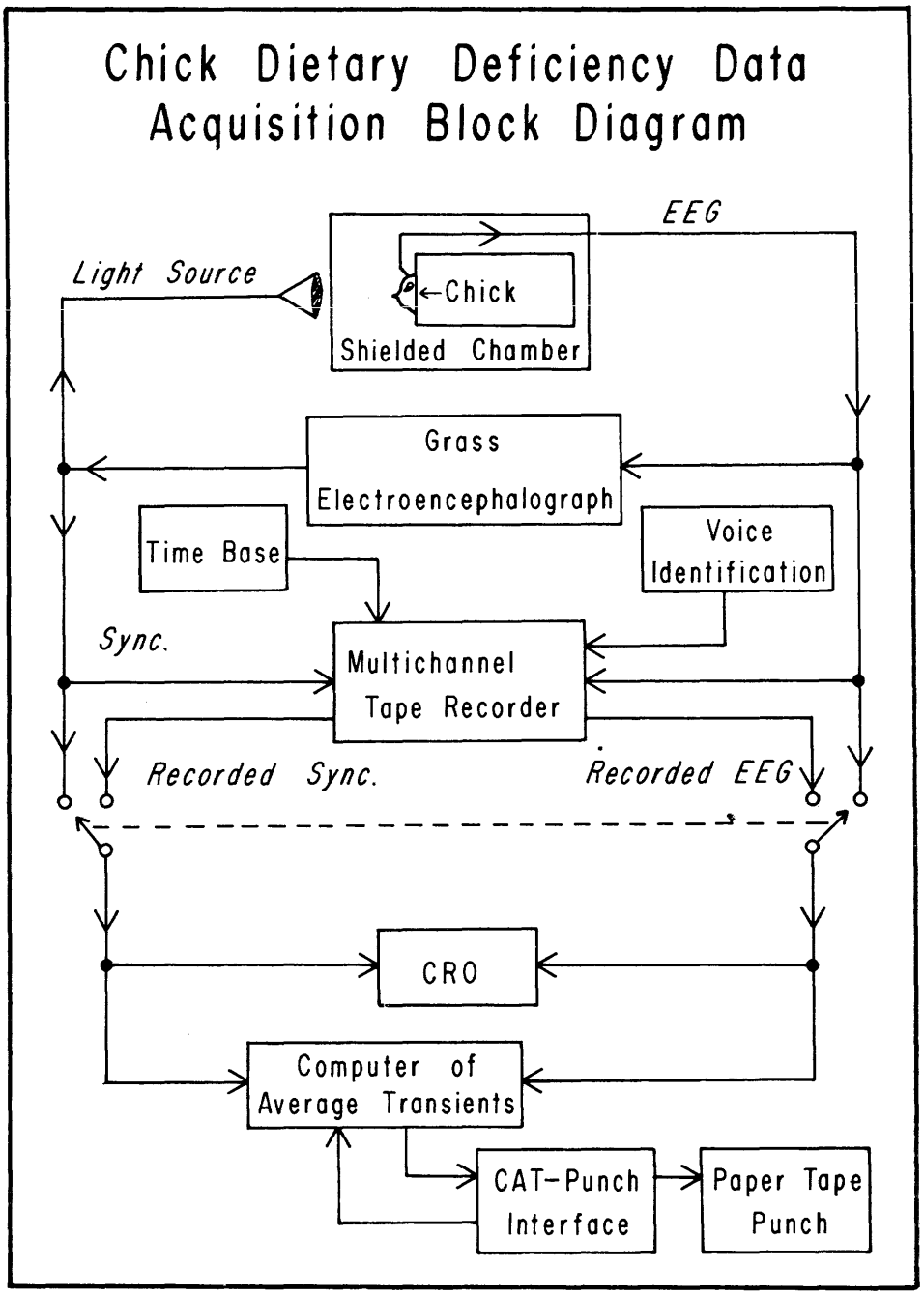


Figure 1 – Chick dietary deficiency data acquisition block diagram

When R=2 the division is termed a dichotomy and categorization consists of determining the sign of a single discriminant function:

$$g(X) \triangleq g_1(X) - g_2(X)$$

The decision rule is as follows:

$g(X) > 0$ --- classify X as belonging to R_1;

$g(X) < 0$ --- classify X as belonging to R_2.

Non-parametric training methods initially assume a form for a discriminant function, an example of which is given as equation (4):

$$g(X) = w_1x_1 + w_2x_2 + \ldots + w_dx_d + w_{d+1} \quad (4)$$

The x_i's are the individual measures which constitute the pattern and the w_i's are adjusted by application of training procedure utilizing a representative group of training patterns.

Parametric classification assumes that each pattern is known a priori to be characterized by a set of parameters, some of which are unknown. Construction of the appropriate discriminant functions incorporates estimation of the parameter values from a representative set of training patterns.

A parametric classification technique has been applied with some success to the chick dietary study while it is anticipated that non-parametric methods

are more applicable in the case of the cough recognition study.

The chick diet deficiency study has as its objective the development of EEG techniques for diagnosis and classification of experimentally induced vitamin B_6 (pyridoxine& deficiency states in the White Leghorn chick. Chicks are initially assigned on a random basis to three groups: control, short term deficient, and long term deficient. The long and short term deficient chicks are placed on a 98% pyridoxine deficient ration on the first and tenth days, respectively.

Electrodes are implanted during an early state of experimentation. Spontaneous and evoked EEG are recorded at 12 hour intervals starting when the chicks are nine days old. The recording sessions continue for two weeks, after which time anatomical histological confirmation of electrode position is made.

Figure 1 is a block diagram of the data acquisition set-up for the study. Chicks are placed in a restraining box in a roosting position. In this position they tend to fall into a partial sleep, which is favorable for recording data. Ten seconds of spontaneous EEG and 200 evoked responses are recorded on magnetic tape. The chick's head is held in position and the left profile is exposed to the stimulating light source. The right eye is covered to insure monocular viewing.

The electrocortigram from the implanted electrodes is recorded on one channel of a multi-channel recorder while identification, time reference and times of occurrence of photic stimuli are recorded simultaneously on other tracks.

During recording the signal is monitored by the CRO and ink recordings are made. The Computer of Average Transients is used in the real-time situation to verify that acceptable data is being recorded.

The Computer of Average Transients contains 400 words of core storage. The memory is stepped sequentially at adjustable rates. The analog signal presented to the input is digitized in synchronization with memory address changes and the digitized value of the signal is added to the data contained in the memory word. When a repetitive signal is presented to the CAT input with an appropriate synchronization signal (in this case the occurrence of the photic event), a considerable enhancement of the signal to noise ratio is obtained.

The memory contents are displayed by stepping the memory sequentially and performing a D/A conversion on the contents of each address. The address register is also subjected to D/A conversion, yielding a voltage which is proportional to the time of occurrence of the averaged amplitude with respect to the synchronizing signal. These converted signals either control a cathode ray tube display which is a part of the CAT, or are available for use by an X-Y plotter.

Provision is additionally made for parallel read out of the memory in BCD form. An interface has been constructed which synchronizes the CAT memory and a paper tape punch so that the digitized data is made available in a form compatible with the SDS 930 computer.

A cough recognition study is currently under way which presents a contrast in data gathering and reduction techniques.

A group is conducting research on respiratory ailments at Southwestern Medical School in Dallas, Texas. One parameter of interest is the number of times that a patient coughs during a given interval. Microphones are placed in selected hospital rooms and audio recordings are made.

Figure 2 is a block diagram of one channel of the data acquisition system. The tape recorders operate in a start-stop mode, having been energized by a voice controlled relay. Sufficient delay is provided to allow the recorders to come up to speed by prerecording on a continuous tape loop. An adjustable band-pass filter is interposed between the amplified signal from the hospital rooms and the voice actuated relay. Preliminary attempts to adjust the filter to allow recording only of coughs has not been successful. Various artifacts will also energize the system when it is adjusted to record the majority of the coughs.

It is therefore necessary to effect a more sophisticated approach to the cough recognition process. The quantity of data, and the impending installation of multichannel continuously operated recorders indicate that a mechanized recognition procedure will be required.

In addition to the requirement to classify the audio signal as cough or non-cough, the physiological model of the cough indicates that decision theory may profitably be applied to the waveform to identify the cough as having emanated from individuals suffering from different broad classifications of respiratory diseases.

It is anticipated that adaptive non-parametric pattern recognition techniques will be applied to the recorded data to effect separation into the pertinent classes.

For both studies outlined a first step in the data reduction procedure is the digitizing of the recorded analog signals and subsequent storage on digital tape. The SDS 930 computer is used for this purpose.

The relatively low maximum frequency content of the EEG recorded in the diet deficiency study allows a conversion rate slow enough that a complete digitized signal segment may be stored in the computer

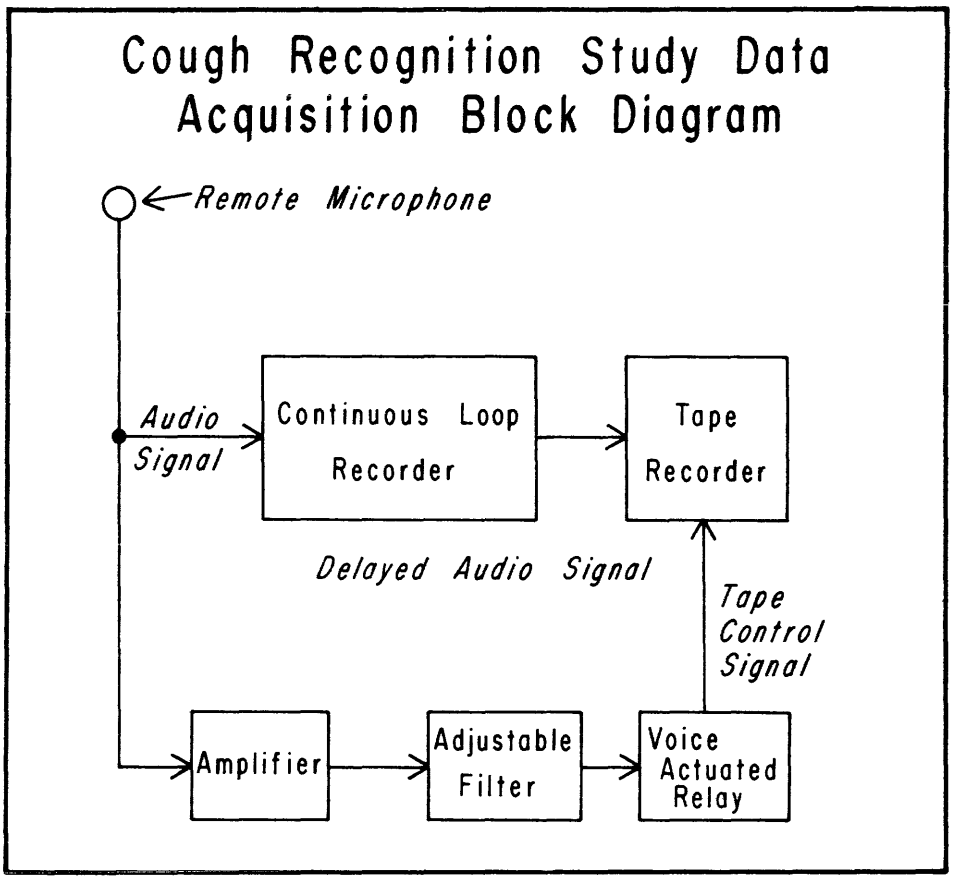


Figure 2 – Cough recognition study data acquisition block diagram

memory prior to writing on tape. In the cough recognition studies the length of an individual signal may be as long as two seconds. At a 20 K conversion per second rate a file consists of some 40,000 conversions, necessitating the simultaneous conversion and writing of information on digital tape. The computer memory acts as a buffer and the central processor controls output format. The priority interrupt system (activated by a pulse generator) initiates a conversion. Sampling time is controlled to within 3.5 microseconds.

Data from the A/D converter is read into the computer memory in the twelve most significant bit positions of a word. The digital words are written in binary mode on tape in a two character per word format, 1000 conversions per record. The two character per word format records data from the 12 most significant bit positions in the memory word.

When the data is read from the tape, it is read in a four character per word mode. This effectively packs two conversions into each memory word.

The final word in a file (a particular signal segment) is an identification word which characterizes the number of records in the file, the number of conversions contained in the file, the sampling rate and the number of conversions in the last data record (all others contain 1000 conversions). File identification is also included. Adjoining files are separated by End of File marks.

Programs have been written to search the tape for a particular file and to output the data contained therein through the D/A converters. Two modes of operation are available – repetitive output of a segment of the file (the maximum length of segment being dependent upon available computer memory) with indexing capabilities so that a whole file may be scanned on a CRO, or continuous output of an entire file for recording on a strip chart recorder. D/A conversion is controlled by the priority interrupt system. The output is therefore adjustable for time base expansion by decreasing the interrupt input pulse rate. The mode of operation is controlled by sense switch operation and the on-line teletype. Provision is made for output of a calibration signal.

Programs have been written for the CDC 6600 to provide the interface necessitated by the difference in word length and internal representation between the two computers.

As previously noted, one of the early decisions that must be made in a pattern recognition study is which measures are significant. The quantity of data that is available in a file is formidable. The two second re-

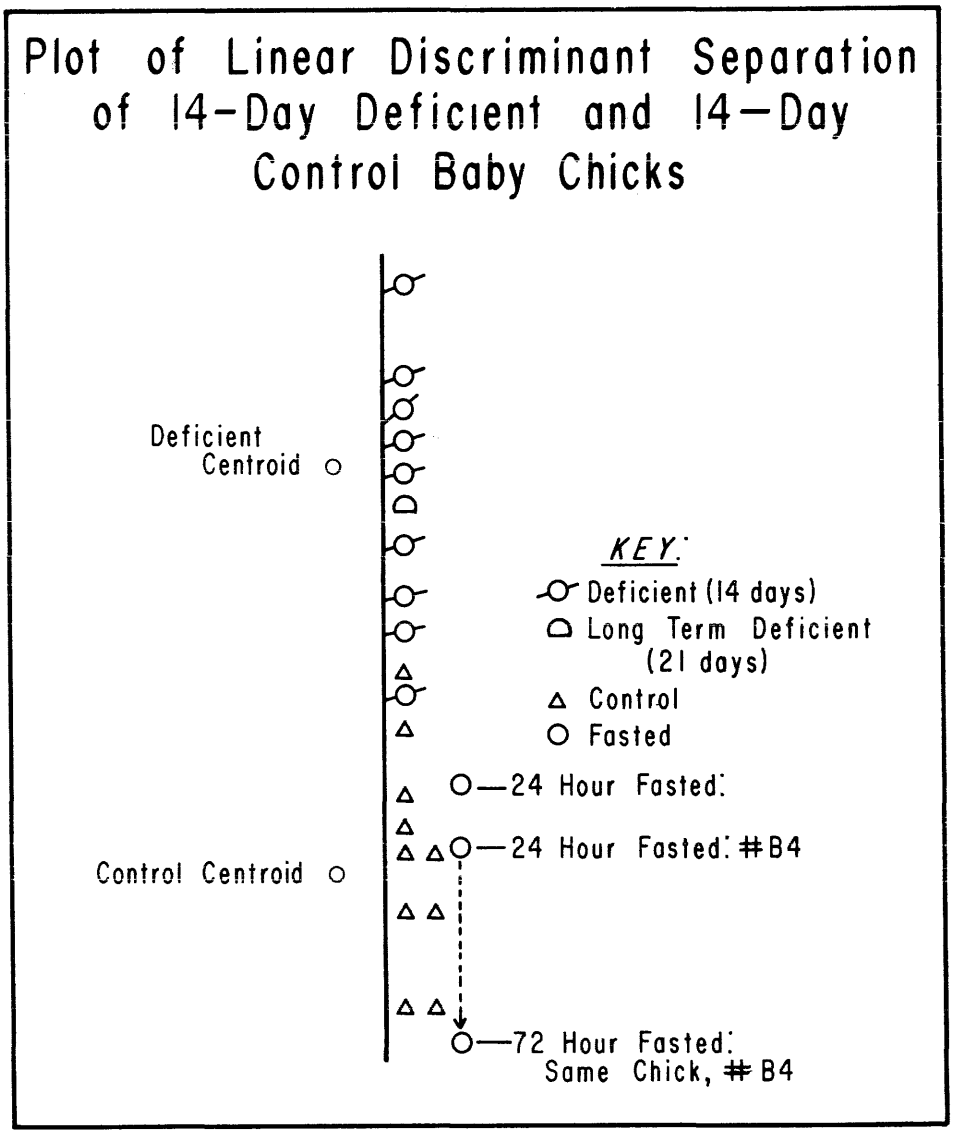


Figure 3 – Plot of linear discriminant separation of 14-day deficient and 14-day control baby chicks

cording mentioned in conjunction with the cough recognition study would have 40,000 data points. If each point is considered as a dimension, direct application of pattern recognition techniques to the sampled amplitude data without pre-processing would necessitate working with 40,000 dimensions. It is necessary, therefore, to find an efficient means to reduce the dimensionality of the pattern without losing the means for classification.

One approach to this data reduction is to perform spectrum analysis of the time signal. Frequency analysis of non-periodic records, such as those with which this paper is concerned, can take several different forms. At one extreme is an evaluation of the Fourier integral, which has as its solution a continuous frequency spectrum for each signal analyzed. At the other extreme is an estimation of the power spectral density of the entire class of signals by statistical techniques. Intermediate between these extremes is the possibility of obtaining a representation of the behavior of the signal in the frequency domain in terms of groups of frequencies. This suggests a bank of band-pass filters (the impulse responses of which are represented digitally and convolved with the digitized experimental data).

Programs are in use which implement these three approaches. The first, a Fourier analysis, is applied in the case of the diet deficiency study where the band of frequencies is relatively narrow. Power spectral density estimates are useful in determining the frequency intervals of interest. The digital filters have been applied to the audio data in the cough recognition study.

In the case of the digital filters, one wishes to cover the frequency band with as few filters as possible without losing significant information. After once

having decided to use filters, counting zero crossings of the output of the filter may be accomplished with little added expense in computation time. The zero crossing count should give an indication of the highest amplitude frequency component present in the filter output, at least on the average.

A normalized tabulation of the periods between zero crossings indicates approximately how the frequency content of the filter output is distributed. The audio signal under consideration may be viewed as a signal modulating some carrier frequency. The detection of the modulating signal would yield the envelope of the signal. This may be done digitally by application of Sterling's approximation from numerical analysis. Differentiation and solving for the time when the derivative of the approximation is zero yields the time of maxima or minima. A re-application of Sterling's formulation yields the interpolated value of the amplitude of the signal at that time. Sterling's approximation attempts to fit a polynomial to the discrete points in the neighborhood of the point of interest. Having evaluated the coefficients of the polynomial, one solves for the value of the polynomial at the point of interest. This formulation has the advantage that relatively few time consuming multiplications are necessary.

Another measure which may find application is the rate at which the maximum energy content varies from one filter output in the bank to another.

Having obtained the measures which appear to be pertinent, the next step is application of training procedures (in the case of non-parametric recognition techniques) or making estimations of the pertinent statistical parameters (in the case of parametric recognition techniques) to obtain the discriminant function described earlier.

In the chick dietary deficiency study a preliminary classification of deficient and control chicks has been obtained by using a parametric linear discriminant function on a fourteen point amplitude distribution. The discriminant function, given as equation (5) below, assumes that the points are normally distributed and that the covariance matrices for each class are equal.

The amplitude measurements were obtained at a time when the control and short term deficient chicks did not have a significant weight difference. All chicks were fourteen days old with the exception of the long term deficient chick, which was 21 days old. A plot of the linear discriminant separation is included as Figure 3. The discriminant function separated these groups at a 0.10 level of significance.

The applicable discriminant function is:

$$g(X)=X^t\Sigma^{-1}(M_1-M_2)-\tfrac{1}{2}M_1^t\Sigma^{-1}M_1+\tfrac{1}{2}M_2^t\Sigma^{-1}M_2+\log(p_1/p_2) \quad (5)$$

where X is the pattern vector for each sample of data, M_i is the mean vector for class i, Σ is the covariance matrix and p(i) is the a priori probability for the ith class. The decision boundary given by setting $g(X) = 0$ is normal to the line segment connecting the transformed means $\Sigma^{-1}M_1$ and $\Sigma^{-1}M_2$. Its point of intersection with this line segment depends upon the constant term:

$$-\tfrac{1}{2}M_1^t\Sigma^{-1}M_1+\tfrac{1}{2}M_2^t\Sigma^{-1}M_2+\log(p_1/p_2) \quad (6)$$

It is to be noted that at the time of writing of this paper the two research projects described are in progress and the results are necessarily incomplete. Although it would have been preferable to include final results of the studies, the outline of the work to date emphasizes that such research would not be feasible without the aid of mechanized data handling and high speed computation.

Experimental investigation of large multilayer linear discriminators*

by W. S. HOLMES and C. E. PHILLIPS

Cornell Aeronautical Laboratory, Inc.
Buffalo, New York

INTRODUCTION

Although proofs of convergence and general expressions for capability have been published to describe the performance of linear discrimination pattern recognizers, very little mathematical basis exists for designing a specific system to perform a desired task. Until these bases are developed, empirical studies are important in order to establish bounds of performance and to find insights that may eventually be generalized into mathematical descriptions of performance.

This study confines itself to perceptrons and parametric modifications to perceptrons. The adaline is included in the study as a perceptron with only one positive connection per A-unit and a threshold, T, subject to $0<T<1.0$. Normally, adalines are fully connected to the input space, that is, there are as many connections (hence A-units) as there are "retinal" points. Truncated adalines in which connections are made to a fraction of the retinal points are also of interest since, for reasonable size input spaces, many retinal points are almost certainly noncontributory.

Normally, a perceptron consists of A-units to which several connections, some positive, some negative, are made from the input space, and in which the connections are selected at random. The "design" of such a system to perform a single dichotomy consists of a statement of the following parameters:

1. Number of A-units.
2. Number of positive connections to each A-unit.
3. Number of negative connections to each A-unit.
4. The threshold value for each A-unit.
5. The threshold value of response units unless it is involved in the adaption policy.

One might include the initial weights coupling the A-units to the response units, the adaption policy, and the training sequence in the "design" since it can be shown that the final recognizer is affected by these factors. In this study, however, these training factors are standardized to as great an extent as feasible and attention is focused on machine parameters.

The experimental results are presented in two parts, one aimed at establishing limits and comparisons of performance for perceptrons, adalines and truncated adalines, for recognition of noisy patterns without transformations; the second to secure information about the generalization capability of such machines when presented with patterns subject to translational transformations.

Experimental methods

The results of pattern recognition experiments are frequently as much influenced by the experimental methods employed as by the nominal techniques being studied.

To perform a recognition experiment employing a multilayer linear discriminator, we first generate a large set of patterns to be classified. These patterns may be purely synthetic or actual photographic images; however, it has been found that the former is experimentally more practical because of the lower cost per sample and the control afforded by known statistical characteristics of the sample set.

Factors that govern the size of the training set required are:

1. Noise.
2. Aspect variations.
3. Translation variations.
4. Rotation variations.

*This paper is based in part on work sponsored by the Air Force Avionics Laboratory, Research and Technology Division, Air Force Systems Command, United States Air Force, under Contract Nos. AF33 (616)-8305 and AF33 (615)-1451. Another portion of this work was sponsored by the United States Navy, Office of Naval Research under Contract No. Nonr 3161(00). The remaining work was performed was performed under Internal Research funds of Cornell Aeronautical Laboratory, Inc.

5. Size variations.
6. Desired classification accuracy.

The more of the above found in a given pattern set, the larger the training set required to resolve the patterns.

A subset of the available patterns is designated as the training set with the remaining patterns forming the test set. From these, a "property" or A-unit catalog is generated. This catalog is a listing of A-unit activities for every pattern in the training and testing set. As such, it is an amalgamation of pattern and system characteristics.

The recognition results to be presented later were obtained through the use of the following pattern sets. The Type 1 patterns consisted of TU104, IL18, LA60, F102, SHIPS, BUILDINGS, and TANKS.

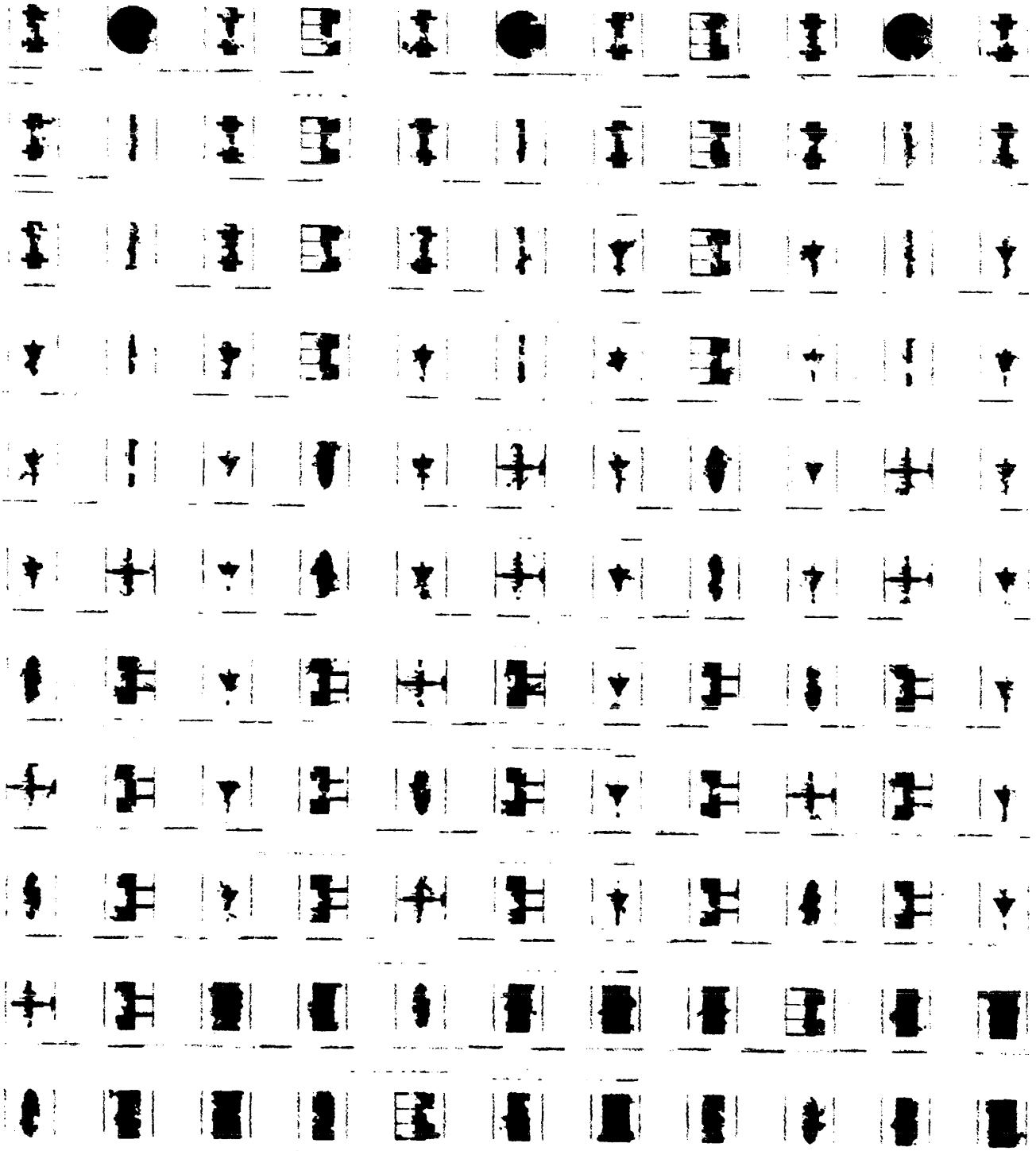


Figure 1—Sample of Type 1 objects used in recognition experiment

versa. When every A-unit is connected to every retinal point with one negative and one positive connection, the results would be the same as for no connections at all. Clearly, between these extremes, some optimum connection density must exist.

Using the Type 1 patterns, a family of perceptrons was trained and tested. As shown in Figure 4, two independent perceptrons were tested at each of seven connection densities ranging from one positive and one negative to 100 positive and 100 negative. We can conclude from the results of this experiment plotted together with 95% confidence limits, that the minimum we expected is extremely flat. Thus, the perceptron appears to be insensitive to number of connections, and the simpler, minimally-connected machines should be preferred over more complicated, many-connection perceptrons.

Since the connection density experiment just presented was based on equal positive and negative connections, and the smallest perceptron tested had two connections, one positive and one negative, it is useful to describe the connection pair for each A-unit as a dipole. The dipole length is the distance from positive connection point to negative connection point in retinal space. Experiments to evaluate sensitivity of dipole length have so far been inconclusive.

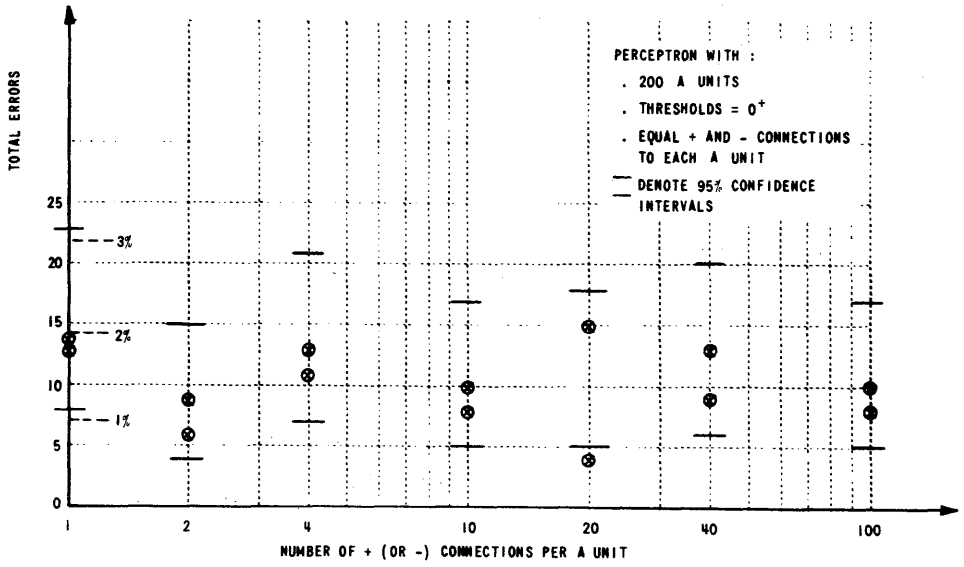


Figure 4—Perceptron performance as a function of number of S-A connections

Adalines

It is difficult to make comparisons of adalines and perceptrons since the performance of the latter is a function of the number of A units equal the number of retinal points.

Nevertheless, using the Type 2 pattern set, the following comparative performances can be observed:

Table I—Comparison of perceptions and adalines

Organization	*Number of A-units*	*Number of connections per A-unit*	*Dichotomy*	*"Retina"*	*% recognition errors*	
					Positive class	*Negative class*
Perceptron	300	2	A/C vs all others	24 × 24	1.0	0.5
Adaline	576	1	A/C vs all others	24 × 24	0.0	0.5

Figure 1 shows a subset of these patterns. Type 2 patterns were F102, IL18, LA60, TU104, two types each of airports, radar sites, missile bases, and command posts. Figure 2 shows a subset of these patterns. Type 3 object patterns were aircraft, artillery, and tanks.

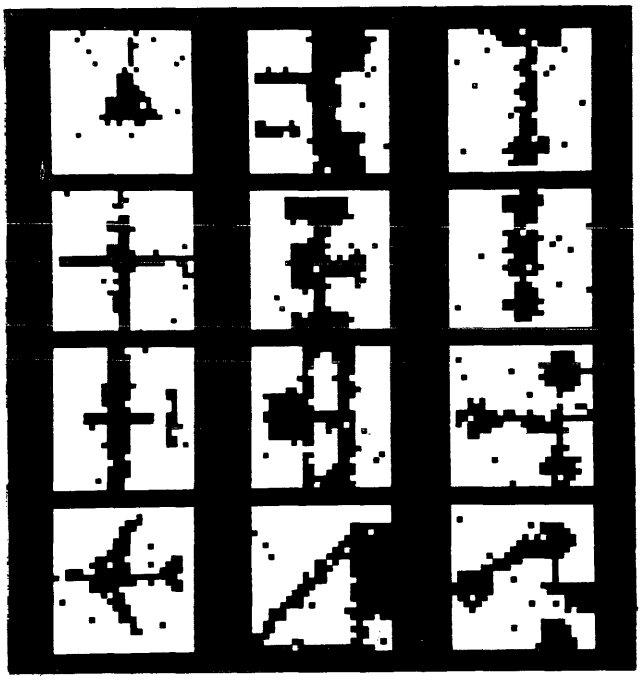


Figure 2—Sample of Type 2 objects used in recognition experiment

System parameter experiments

Traditional perceptions

In order to measure the sensitivity of N_A, the number of A-units, the Type 1 pattern set was used in a single dichotomy experiment placing all aircraft in the X-class and all other patterns in the Not-X-class. There were 40 connections to each A-unit from the input space except for two machines (100 A-units and 200 A-units) which had 20 connections per A-unit. The connections were equally divided between positive and negative polarities and the A-unit thresholds were zero.

Now, this experiment requires that a number of perceptrons be constructed, trained, and tested. Each point in Figure 3 represents the performance of a different machine, each of which is parametrically alike except in the number of A-units. The figure shows an entirely expected result, that as the number of A-units in a machine is reduced performance deteriorates and that below some number of A-units, in this case 100, performance deteriorates rapidly. This knee of the curve must certainly be a function of the "difficulty" of the problem in some sense.

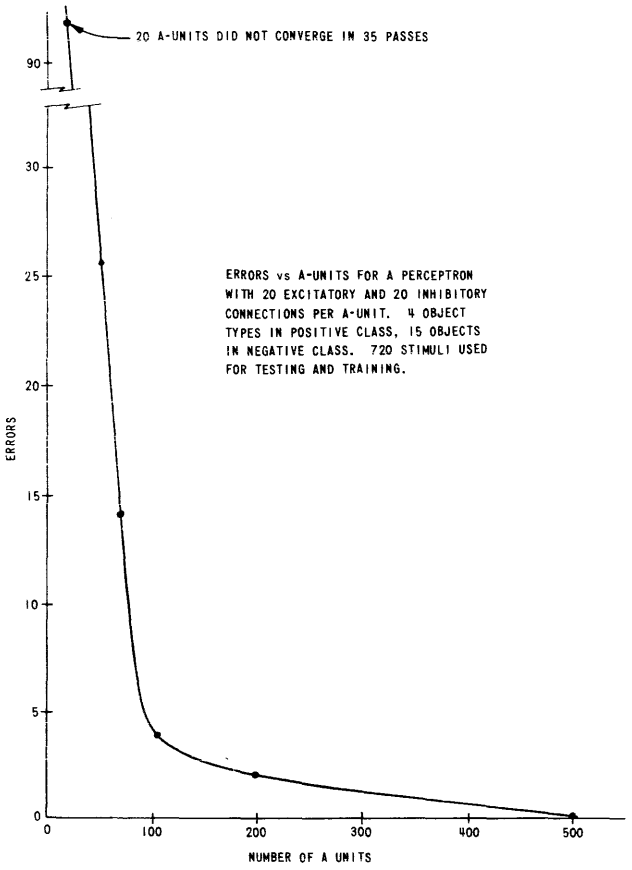


Figure 3—Perceptron performance sensitivity to number of A-units

Since in References 4 and 5, it is shown that, by rational point-set selection, very much smaller machines, i.e., fewer numbers of A-units, can perform as well as random point set machines, an order of magnitude larger, we may suspect that some randomly selected point sets may be better than others for a given problem. Thus, for truly reliable results, this experiment should be replicated many times using a random selection process independent from one machine generation to the next.

A more significant question than performance as a function of number of A-units relates to the number of connections per A-unit. If we have equal numbers of positive and negative connections, keep A-unit thresholds at zero, but vary the number of connections per A-unit, recognition performance should minimize somewhere between zero connections and fully connected. At zero connections, the system would be unresponsive to patterns and either erroneously recognize all X-class patterns as Not-X-class or vice

The adaline which has about twice the number of A-units as the perceptron, demonstrates a performance roughly three times better than the perceptron. Other experiments have shown that zero-information A-units can, through participation in the training process, induce a solution inferior to that resulting when they were eliminated from the training process. These observations lead to a suggestion that, for reasonable size retinas, truncated adalines should be explored.

Truncated adalines

Elimination of connections to the retinal space reduces the size of the system, and hence its cost. Elimination on the basis of an information analysis of the training set permits a drastic reduction as shown in Table II. The Type 2 pattern set was used in these experiments.

Table II—Truncated adalines

Number of A-Units	% Recognition		Overall	Truncation %
	Posiitve Class	Negative Class		
576	100	99.5	99.7	0
191	98.5	99.0	98.3	67
99	93.5	98.1	96.6	83
48	82.5	95.6	91.3	92

Although performance suffers some from truncation, the deterioration is small compared to system savings in terms of retinal units and weights in the linear discrimination function of the system.

Pattern translation experiments

General

Noise presents a significant test of a recognition system's ability to generalize. Performance in the face of linear transformations of patterns such as translation, rotation, and scale changes presents a conceptually different test of a system's ability to generalize. For example, when considering generalization over translation, it is pertinent to inquire how performance deteriorates as tests are made using patterns located farther and farther from the location of the training set. Further, if the training set is divided between two locations separated by one or more retinal points, will the response to the testing set be better in the bracketed locations than in correspondingly distant locations outside the training bracket? Two translational experiments were performed seeking answers to these and other generalization questions.

The experiments to investigate the translation transformational generalization capabilities of this class of recognition system were all performed using Type 2 patterns. Thus, all patterns in the training and testing set included noise and the sets were large enough to assure statistical significance, at least with respect to the specific machines implemented.

Performance under translational training

This experiment was intended to answer the question of the performance of two-connection perceptrons when the training set is subjected to statistically controlled translations. The initial pattern set was translated under a procedure in which the amount of translation in each pattern was selected from a Gaussian distribution independent in $|x|$ and $|y|$. The testing set was similarly translated and the test results plotted, as shown in Figure 5, as a function of the total translation calculated as follows:

$$T = |x| + |y| .$$

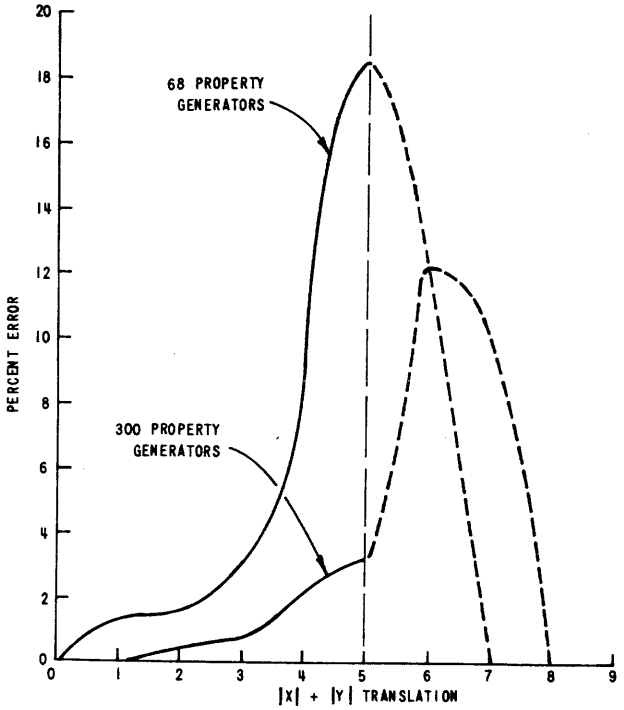


Figure 5—Recognition error vs total translation

This experiment would appear to predict deteriorating performance as a function of increasing translation, at least within the range of reasonable statistical significance. However, when one considers the nature of the translational process, both for training and testing, which would tend to concentrate patterns at certain translational distances, and observes that the number of possible locations for patterns increases with increasing T, it is clear that the data require closer examination. Table III shows the results of the experiment for 68 A-unit and 300 A-unit two-connection perceptrons together with information about the number of patterns involved in the experiment in each translational distance. Focusing attention on the column "patterns per location," which is based on the

total number of patterns at a fixed value of T and the number of locations corresponding to that value of T, we can see that this experiment tends simply to reveal that the performance of the two perceptrons tested deteriorates as the number of patterns per location reduced. Although the data are a little sparse for use in drawing general conclusions, Figure 6 shows that they are suggestive of an inverse relationship in number of patterns and that upwards of 50 to 100 patterns per location would be required to effectively train such a device for translational efficiency. If one econsiders a 100 × 100 retinal region, the problem of training a machine using adequate samples at each of the 10^4 locations becomes startling.

Table III—Recognition under training and testing translations

A		B		C		C/B	
|x| + |y| Translation	Number of locations	Total number of patterns in set with translation as given in column A	Average patterns per location	Number of patterns misclassified (Two connections) 68 A-units	200 A-units	% Error 68 A-units	300 A-units
0	1	86	86	0	0	0.0	0.0
1	4	289	72	4	0	1.384	0.0
2	8	345	43	5	2	1.449	0.579
3	12	259	21	8	2	3.088	0.772
4	16	133	8	13	3	9.774	2.255
5	20	59	3	11	2	18.664	3.389
6	24	16	0.6	2	2	12.5	12.5
7	28	10	0.4	0	1	0.0	10.0
8	32	3	0.1	0	0	0.0	0.0

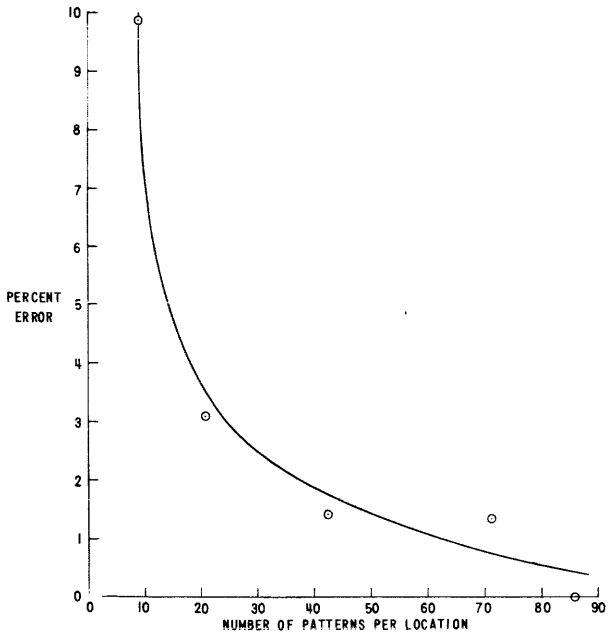


Figure 6—Relation between recognition performance and number of samples per translation

Translational generalization

Since the pattern set required to train a perceptron adaline with two-translational transforms over a reasonable size retinal space is prohibitive, an experiment was run to determine the ability of such recognition systems to recognize patterns in locations adjacent to the location of the training set. For this purpose, the Type 3 pattern set was used. This set has 880 training patterns located at positions zero, +2, and −2 resolution cells. Testing subsets are available with translations zero, +1, +2, +3, and +4. The testing subset translated +1 resolution cell is bracketed by the 0 and +2 training subset locations. Testing subsets translated by +3 and +4 are unbracketed by any subsets and are progressively more remote from any location at which training has taken place. It is also of interest to examine the behavior of the machine at 0 and +2 locations in order to determine whether the influence of the training at +2 and −2 has caused the machine to be more capable as a recognizer at the 0 point than at the +2 point. The data from this experiment are presented graphically in Figure 7, plotting the number of errors as a function of translation of the test set. The letter T in the diagram designates the locations at which training took place.

The data contain one mystery which needs resolution, but in general, suggests that a limited amount of translational generalization is taking place. The mystery relates to the behavior of the uniform random point-set selection device in its performance beyond +2. Whereas we would expect the performance to deteriorate and, in fact, to deteriorate faster than for the +1 translational point, it, in fact, appears to improve. Performance at 0 is better than at +2 both of which

were subjected to training, the first of which, however, was bracketed by +2 and −2.

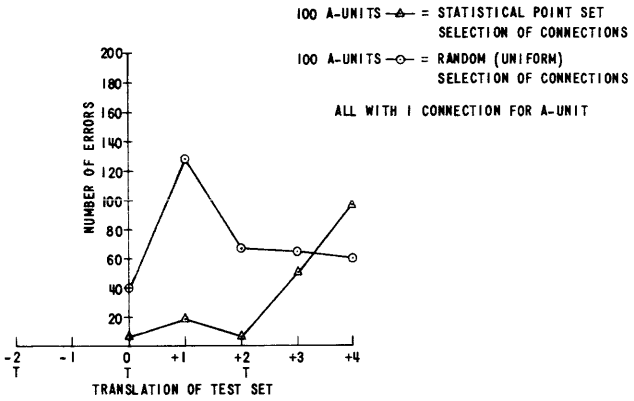


Figure 7—Two experiment example of translational generalization

As mentioned in References 4 and 5, statistical point set selection of connections tends to produce a machine which is "tuned" to the problem. We had anticipated that the performance of the statistical point set selection machine would be relatively poorer in the +1 testing location than the random selection machine. This conjecture is not borne out by the data. Therefore, we must conclude that the statistical point set selection process, while tending to "tune" to the data, still permits significant generalization relative to random point-set selection machines. This relatively good generalization probably arises because of the great waste of connections in the randomly selected machine.

This experiment suggests that a machine having strong generalization capabilities might be constructed by training in a matrix of resolution cells which are two or three units separated from each other in every direction. This would cut down the cost of training such a device by a factor of 4 to 9. An experiment to confirm this speculation has not yet been performed so it is not possible to go beyond the speculation.

SUMMARY AND CONCLUSIONS

The experiments described in this paper suggest that the number of connections per A-unit in perceptrons should be reduced to the greatest extent possible and the reduction to one connection per A-unit seems economically optimum as long as statistical point set selection techniques are employed. The size of training sets must be determined by the number of discrete points of transformation to be encompassed by the final machine and the distribution of training points over the transformations need not be as fine as the distribution of resolution cells in the "retinal" space. Even with this relaxation in requirements upon training, the cost of training a recognition system over more than one transformation, for example, translation, would probably be prohibitive. Therefore, additional constraints and/or organizational modifications to the systems tested seem required in order to achieve economic recognition efficiency over a sizable spectrum of transformations of the pattern set.

ACKNOWLEDGMENTS

The contributions of C. W. Swonger, J. B. Beach, and C. T. Phillips to the analysis of property information relationships are acknowledged. The Type 1 patterns employed in the reported recognition experiments were generated under Contract No. Nonr 3161(00), supported by the Geography Branch of the Office of Naval Research. Generation of the Type 2 patterns and performance of some of the experiments in this work were accomplished under Contract No. AF33(615)-1451 supported by the Research and Technology Division, Wright-Patterson Aif Force Base.

REFERENCES

1 W. S. HOLMES, H. R. LELAND, and G. E. RICHMOND
Design of a photo-interpretation automaton
presented at the Fall Joint Computer Conference 4 December 1962

2 F. ROSENBLATT
Principles of neurodynamics
Cornel Aeronautical Laboratory, Inc. Report No. 1196-G-8 (also published by Spartan Books, Washington, D. C. 15 March 1961

3 W. S. HOLMES, H. R. LELAND and J. L. MUERLE
Recognition of mixed-font imperfect characters
presented at Optical Character Recognition Symposium Washington, D. C. 16 January 1962

4 M. G. SPOONER, C. W. SWONGER, and J. B. BEACH
An evaluation of certain property formation techniques for pattern recognition
presented at WESCON, Los Angeles, California 25-27 August 1964

5 C. W. SWONGER and C. T. PHILLIPS
A report of property generation and utilization research for pattern recognition
Cornell Aeronautical Laboratory, Inc. Report No. VG-1860-X April 1965

A computer analysis of pressure and flow in a stenotic artery*

by PETER M. GUIDA, *M.D.*
The New York Hospital-Cornell Medical Center
New York, New York

LOUIS D. GOLD, *Ph.D.*
Electronic Associates, Inc.
Princeton, New Jersey

and

S. W. MOORE, *M.D.*
The New York Hospital-Cornell Medical Center
New York, New York

INTRODUCTION

There is a widespread assumption in the medical and engineering communities that when a small decrease in cross-sectional area is imposed upon an artery there is a correspondingly small decrease in arterial flow and pressure, and that when progressively larger increases in arterial stenosis are made there is a correspondingly greater decrease in arterial pressure and flow. That this is not so has been shown in earlier studies[1,2] in a qualitative way. When a small stenosis is imposed upon an artery there is no change in arterial pressure and flow until a rather large and significant stenosis is produced. Recent studies[3,4] have attempted to develop this concept even further. It has been the object of this study to quantitate the effect on the pressure and flow in an artery that is subjected to an increasing stenosis. This has been done by drawing from a background of numerous arterial flow and pressure determinations in both humans and dogs, and then carrying this further to the development of a mathematical model and subjecting the mathematical model to computer analysis.

*Portion of this work was done during a tenure by Dr. P. M. Guida as an Advanced Research Fellow of the American Heart Association. This work was supported by USPH Grant # HEO6846, the Susan Greenwall Fund and the Charles Washington Merrill Fund.

Materials and methods

The basis for the construction of the mathematical model was determined in the following manner: Numerous blood flow measurements were made with a square-wave electromagnetic blood flowmeter.* Pressure measurements were made through No. 20-gage needles inserted into the artery and connected to short lengths of rigid wall nylon tubing which, in turn, were connected to strain gages.** These were connected to strain gage pre-amplifiers† and displayed on a suitable oscilloscope.†† The data were recorded on a 7-channel instrumentation tape recorder§ arranged in the FM mode of recording with a frequency response flat from 0 to 2500 Hz. Photographs were made of the analog oscilloscope display by using a Tektronix Model C-12 Polaroid Camera that can take photographs directly from the face of the oscilloscope tube. These data were then subjected to qualitative analysis, reduced manually to digital form, and the mathematical model which is described here was developed. The requisite mathematical

*Avionics Research Products, Model MS-6000A.
**Statham Model P-23Db.
†Tektronix Model Q Oscilloscope Plug-in.
††Tektronix Model RM-535A Oscilloscope.
§Ampex Model SP-300 Instrumentation Tape Recorder.

equations were solved and the results plotted out on a Digital Computer.§§ The Computer Block Diagram is presented in Figure 1.

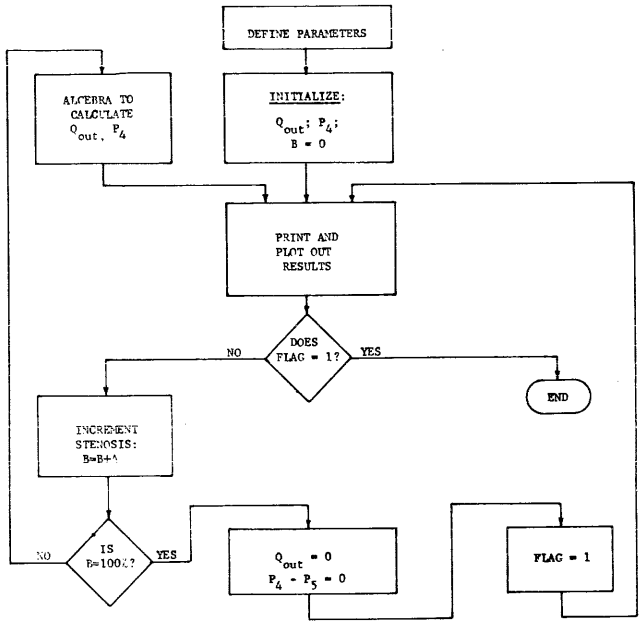


Figure 1—Computer block diagram for computation of distal flow and pressure in a stenotic artery

Mathematical model

In developing a mathematical model of the portion of an artery in stenosis, we will idealize the artery to the extent of considering only rigid wall tubing of constant cross-section, except in the immediate area of the stenosis. This idealized system is shown in Figure 2.

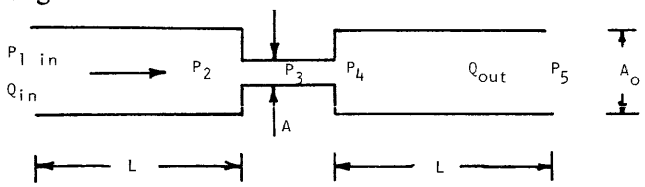


Figure 2—Schematic representation of an idealized stenosis in an artery

A momentum balance between points 1 and 2 and points 4 and 5 yields:

$$\frac{dQ_{in}}{dt} = \frac{k_1 A_o}{pL}(P_1 - P_2) - \frac{RQ_{in}}{pA_o} \tag{1}$$

$$\frac{dQ_{out}}{dt} = \frac{k_1 A_o}{pL}(P_4 - P_5) - \frac{RQ_{out}}{pA_o} \tag{2}$$

where Q is the volumetric flow rate, R is the resistance to flow, and p is the density of blood. Under these conditions, the continuity equation takes the form:

$$Q_{out} = Q_{in}\ (p = \text{constant}) \tag{3}$$

therefore, by virtue of equation (3) we make equate equations (1) and (2):

$$\frac{dQ_{in}}{dt} = \frac{dQ_{out}}{dt};\ \text{or} \tag{4}$$

$$P_1 - P_2 = P_4 - P_5$$

Application of the Steady State Bernoulli equation (i.e., Conservation of Energy) to section 2, 3 in Figure 2 yields:

$$\Delta(1/2v^2/k_1) + \frac{\Delta P}{p} + 1/2\,(v_3)^2\, e_v = 0 \tag{5}$$

where v is the linear velocity, e_v is the total energy loss in the section under consideration, and k_1 is a conversion factor. Then:

$$\frac{v_3^2}{2k_1} - \frac{Ev^2}{2k_1} + \frac{P_3 - P_2}{p} + 1/2\, v_3^2 \frac{e_v}{k_1} = 0 \tag{6}$$

We may evaluate e_v for a sudden contraction[5] as:

$$e_v = .45\ (1\text{-}B) \tag{7}$$

where

$$B = A/A_o;\ \text{thus,}$$

$$v_3 A = vA_o$$

or

$$v_3 = v\frac{A_o}{A} = v/B$$

Therefore, (6) becomes:

$$\frac{v^2}{2B^2k_1} - \frac{v^2}{2k_1} + \frac{P_3 - P_2}{p} + 1/2\,\frac{v^2}{B^2k_1}(.45)\,(1\text{-}B) = 0 \tag{8}$$

or

$$P_3 = -\frac{pv^2}{2k_1}\left[1/B^2 - 1 + \frac{.45}{B^2}(1 - B)\right] + P_2 \tag{9}$$

§§Electronic Associates Inc. Model 8400 Digital Computer.

Similarly, applying Bernoulli's Equation to sections 3, 4 in Figure 2 yields:

$$P_3 = \frac{p}{2k_1} v^2 [1 - 1/B^2 + (1/B - 1)^2] + P_4 \tag{10}$$

since e_v for a sudden expansion is [6]: (11)

$$e_v = (1/B - 1)^2$$

Equating equations (9) and (10) gives:

$$P_2 = P_4 + \frac{pv^2}{2k_1}(1/B - 1)^2 + .45\,\frac{(1 - B)}{B^2} \tag{12}$$

We may substitute the continuity relationship:

$$P_1 - P_2 = P_4 - P_5$$

into equation (12) to give:

$$P_2 = \frac{P_1}{2} + \frac{P_5}{2} + \frac{pv^2}{4k_1B^2}[(1 - B)^2 + .45\,(1 - B)] \tag{13}$$

Substituting (13) into equation (1) yields:

$$\frac{dQ_{in}}{dt} = \frac{k_1 A}{pL}\left\{\frac{P_1}{2} - \frac{P_5}{2} - \frac{pv^2}{4k_1B^2}\right.$$

$$\left.[(1 - B)^2 + .45\,(1 - B)]\right\} - \frac{RQ_{in}}{pA} \tag{14}$$

Now, analysis of the actual experimental data and the numerical values of the parameters in equation (14) implies that there is no phase-lag in the pressure-flow relationship, or:

$$\frac{dQ_{in}}{dt} = 0$$

Thus, we finally obtain from equation (14) the desired relationship:

$$Q^2\left\{\frac{(1 - B)^2}{2B^2} + \frac{.45}{2}\,\frac{(1 - B)}{B^2}\right\}$$

$$+ \frac{2RLQ}{p} - \frac{k_1 A_o^2}{p}(P_1 - P_5) = 0 \tag{15}$$

from which the flow rate, Q, may be determined.

We may also determine the distal pressure, P_4, by considering equation (2) and noting that

$$\frac{dQ_{out}}{dt} = 0$$

or

$$\frac{dQ_{out}}{dt} = 0 = \frac{k_1 A_o}{pL}(P_4 - P_5) - \frac{RQ_{out}}{pA_o} \tag{16}$$

Therefore

$$P_4 = P_5 + \frac{RL}{k_1 A_o^2} Q_{out} \tag{17}$$

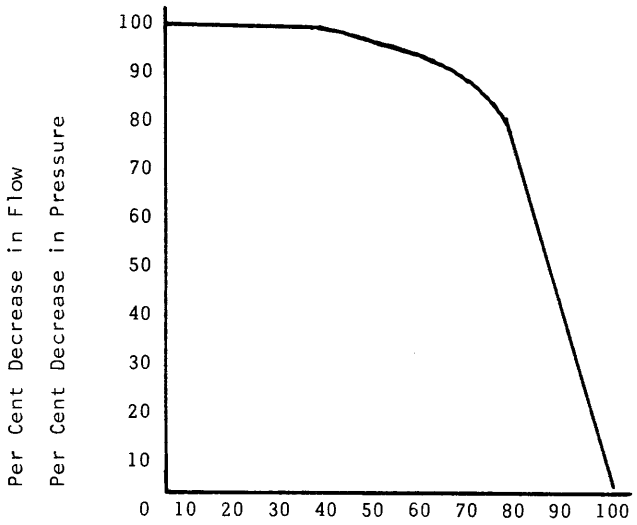


Figure 3 – A plot of the computer results of the effect of increasing arterial stenosis upon flow and pressure. Only one curve is shown since the flow and pressure curves are identical

Results and discussion

Figure 3 presents the solutions to equations (15) and (17) as plots of the distal flow, as a percentage of the normal-unobstructed-flow, and the distal pressure difference, $P_4 - P_5$, as a percentage of the normal pressure difference, against increasing stenosis in the artery.

Figure 4 is a cross-plot of the percentage of unobstructed distal flow against the percentage of unobstructed distal pressure differential, with stenosis as an implicit parameter. Note from Figure 3 the point of critical stenosis at 75 per cent (i.e., where flow and pressure drop precipitously), and from Figure 4 the expected linearity of the pressure-flow relationship, independent of stenosis, as predicted by equation (15).

Figure 5, a plot of flow and pressure against steno-

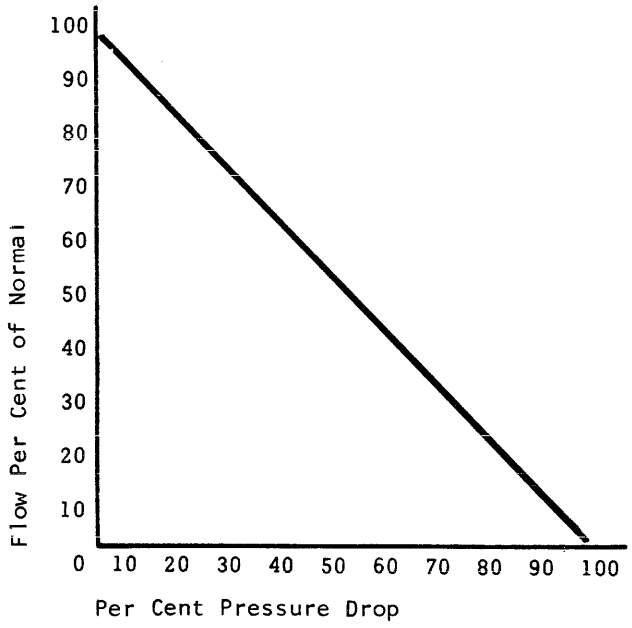


Figure 4—Plot of pressure drop and flow in an increasingly stenotic artery

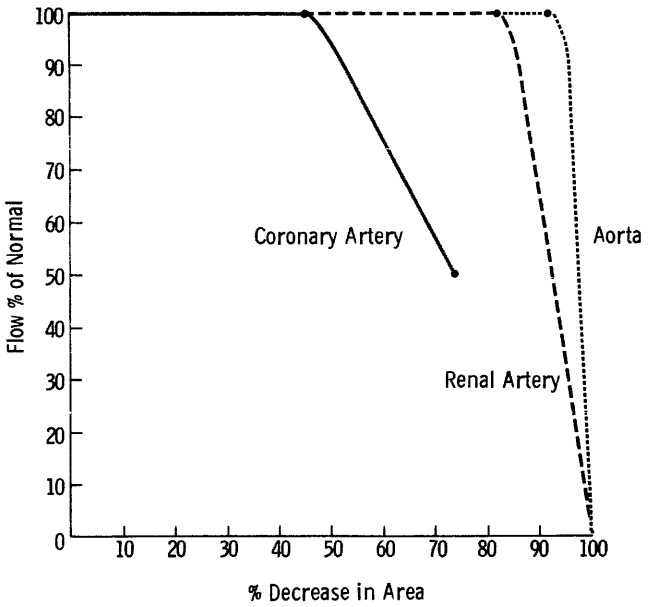


Figure 5—Plot of actual in vivo measurements of the effect of increasing arterial stenosis upon blood flow. The point of critical decrease in cross-sectional area in the descending thoracic aorta is 92 per cent; in the renal artery, 82 per cent; in the anterior descending branch of the left coronary artery, 45 per cent

sis, is derived from actual in vivo experimental measurements. These curves are presented to illustrate that there are different values of critical stenosis for different arteries. The lowest extreme value is the anterior descending branch of the left coronary artery where flow and pressure begin to fall off precipitously at 45 per cent decrease in cross-sectional area. For the descending thoracic portion of the aorta this value is 92 per cent. The renal artery plot is shown between these two extremes. The value for this artery is 82 per cent. The lower value for the coronary artery is due, in part, to a lower, driving pressure head (diastolic pressure) and occurs at a later time in the cardiac dynamic cycle. The parameters chosen for the mathematical model, (e.g., L, A_o, etc.), are not the same, however, as those implied in Figure 5. The two curves are quite similar, supporting the mathematical model postulated here. Nevertheless, discrepancies between the present results and actual physical responses may be expected in regions of very large stenosis (almost 100 per cent obstruction), since then the assumption of negligible phase-lag (i.e., $\frac{dQ}{dt} = 0$) is likely to be in error. Indeed, the very reason for this error is that in such situations the elasticity of the arterial walls must be considered. The effect of elasticity will cause the continuity relation:

$$Q_{out} = Q_{in}$$

to break down. Subsequent work will show how the effect of elasticity at large stenosis may be determined. This will lead to a method for determining arterial elastance under actual flow conditions by simple measurements of flow and pressure.

SUMMARY

It had been a previous, widespread assumption that when an artery is subjected to a gradual, progressive, stenosis there was an immediate, gradual, progressive, fall in arterial blood pressure and flow, i.e., the relationship was linear. Experimental work over the past seven years has shown that this is not so; the relationship is not linear. Experimental observations in the human and on the dog have shown that when an artery is subjected to a decreasing cross-sectional area, as produced by a stenosis, the flow through that artery remains unchanged until a severe decrease in cross-sectional area is attained. Prior to this point there is no change in blood flow, i.e., it remains normal. When this point of critical stenosis is reached, there is a sudden, marked, and precipitous drop in blood flow through the artery.

Application of the principles of conservation of energy and momentum to the system under consideration have led to a mathematical system which accurately describes the phenomenon. A computer program has been written which describes this system. The print-out of the computer results is presented in Table 1.

It is further expected that this mathematical system will enable one to determine experimentally the

STENOSIS	QOUT	QPERCT	POUT	PPERCT
0.0500E 00	0.2745600E 03	0.1000000E 03	0.7710000E 03	0.1000000E 03
0.5000E 01	0.2743463E 03	0.9992213E 02	0.7709913E 03	0.9992120E 02
0.1000E 02	0.2741232E 03	0.9984090E 02	0.7709824E 03	0.9984019E 02
0.1500E 02	0.2737759E 03	0.9971440E 02	0.7709685E 03	0.9971368E 02
0.2000E 02	0.2732889E 03	0.9953703E 02	0.7709490E 03	0.9953613E 02
0.2500E 02	0.2726310E 03	0.9929741E 02	0.7709226E 03	0.9929642E 02
0.3000E 02	0.2717316E 03	0.9896982E 02	0.7708866E 03	0.9896906E 02
0.3500E 02	0.2705174E 03	0.9852759E 02	0.7708380E 03	0.9852737E 02
0.4000E 02	0.2688696E 03	0.9792741E 02	0.7707720E 03	0.9792702E 02
0.4500E 02	0.2666278E 03	0.9711093E 02	0.7706821E 03	0.9711026E 02
0.5000E 02	0.2635613E 03	0.9599403E 02	0.7705593E 03	0.9599387E 02
0.5500E 02	0.2593373E 03	0.9445558E 02	0.7703900E 03	0.9445467E 02
0.6000E 02	0.2534776E 03	0.9232137E 02	0.7701553E 03	0.9232066E 02
0.6500E 02	0.2452940E 03	0.8934076E 02	0.7698274E 03	0.8933992E 02
0.7000E 02	0.2338121E 03	0.8515881E 02	0.7693674E 03	0.8515846E 02
0.7500E 02	0.2176922E 03	0.7928763E 02	0.7687216E 03	0.7928688E 02
0.8000E 02	0.1951885E 03	0.7109138E 02	0.7678199E 03	0.7109041E 02
0.8500E 02	0.1642078E 03	0.5980759E 02	0.7665787E 03	0.5980668E 02
0.9000E 02	0.1225357E 03	0.4462983E 02	0.7649092E 03	0.4462891E 02
0.9500E 02	0.6822295E 02	0.2484810E 02	0.7627335E 03	0.2484797E 02
0.1000E 03	0.0000000E 00	0.0000000E 00	0.7600000E 03	0.0000000E 00

Table 1—Tabulation of computer results of the mathematical model simulating the effect of stenosis upon blood flow and pressure through an artery. Note that with increasing stenosis there is practically no change in flow or pressure until the arterial cross-sectional area is reduced 75 per cent

elastic properties of arterial walls under actual flow conditions by measurement of just flow and pressure.

CONCLUSIONS

Numerous determinations of arterial flow and pressure have been made in the human and in the dog in normal arteries and in arteries that have been subjected to stenosis, either by arteriosclerosis or by physical stenosis produced by a circumferential ligature about an artery. Data collected from this source have shown that when a stenosis is imposed upon an artery, there is initially no change in pressure or flow within that artery, proximal to the stenosis, until a certain critical point is reached where arterial flow and pressure simultaneously fall precipitously. This point of critical arterial stenosis is different for different arteries.

Drawing upon a large experimental background, a mathematical model has been developed to define the effect of a gradual and progressive arterial stenosis upon arterial flow and pressure.

Having once defined the mathematical model, the computer model was developed. The computer results were shown to be identical with the actual measurements made on the human and on the dog.

A recheck of the computer model results was then done on the dog in several arteries, and the results in the in vivo preparation corroborated the computer results.

A mathematical and computer model have been developed as an approach to the study of the effect upon arterial pressure and flow when that artery is subjected to a progressive stenosis.

REFERENCES

1 F C MANN J F HERRICK H E ESSEX E J BALDES
The effect on the blood flow of decreasing the lumen of a blood vessel
Surgery 4:249-252 1938

2 T C GUPTA C J WIGGERS
Basic hemodynamic changes produced by aortic coarctation of different degrees
Circulation 3:17-31 1951

3 A G MAY J A DE WEESE C G ROB
Hemodynamic effects of arterial stenosis
Surgery 53:513-524 1963

4 A G MAY L VAN DE BERG J A DE WEESE C G ROB
Critical arterial stenosis
Surgery 54:250-259 1963

5 R B BIRD W E STEWART E N LIGHTFOOT
Transport Phenomena
John Wiley & Sons Chapter 8 New York 1960

6 R B BIRD W E STEWART E N LIGHTFOOT
Transport Phenomena
John Wiley & Sons Chapter 8 New York 1960

Chairman's Introduction to the SJCC Session

Security and privacy in computer systems

by WILLIS H. WARE
The RAND Corporation
Santa Monica, California

INTRODUCTION

Information leakage in a resource-sharing computer system

With the advent of computer systems which share the resources of the configuration among several users or several problems, there is the risk that information from one user (or computer program) will be coupled to another user (or program). In many cases, the information in question will bear a military classification or be sensitive for some reason, and safeguards must be provided to guard against the leakage of information. This session is concerned with accidents or deliberate attempts which divulge computer-resident information to unauthorized parties.

Espionage attempts to obtain military or defense information regularly appear in the news. Computer systems are now widely used in military and defense installations, and deliberate attempts to penetrate such computer systems must be anticipated. There can be no doubt that safeguards must be conceived which will protect the information in such computer systems. There is a corresponding situation in the industrial world. Much business information is company-confidential because it relates to proprietary processes or technology, or to the success, failure, or state-of-health of the company. One can imagine a circumstance in which it would be profitable for one company to mount an industrial espionage attack against the computer system of a competitor. Similarly, one can imagine scenarios in which confidential information on individuals which is kept within a computer is potentially profitable to a party not authorized to have the information. Hence, we can expect that penetrations will be attempted against computer systems which contain non-military information.

This session will not debate the existence of espionage attempts against resource-sharing systems. Rather, it is assumed that the problem exists, at least in principle if not in fact, and our papers will be devoted to discussing technological aspects of the problem and possible approaches to safeguards.

First of all, clarification of terminology is in order. For the military or defense situation, the jargon is well established. We speak of "classified information," "military security," and "secure computer installations." There are rules and regulations governing the use and divulgence of military-classified information, and we need not dwell further on the issue. In the non-military area, terminology is not established. The phrase "industrial security" includes such things as protecting proprietary designs and business information; but it also covers the physical protection of plants and facilities. For our purposes, the term is too broad. In most circles, the problem which will concern us is being called the "privacy problem."

The words "private" and "privacy" are normally associated with an individual in a personal sense, but *Webster's Third New International Dictionary* also provides the following definitions:

> Private:... intended for or restricted to the use of a particular person, or group, or class of persons; not freely available to the public
>
> Privacy:... isolation, seclusion, or freedom from unauthorized oversight or observation.

We are talking about restricting information within a computer for the use of a specified group of persons; we do not want the information freely available to the public. We want to isolate the information from unauthorized observation. Hence, the terminology appears appropriate enough, although one might hope that new terms will be found that do not already have strongly established connotations. For our purposes today, "security" and "classified"

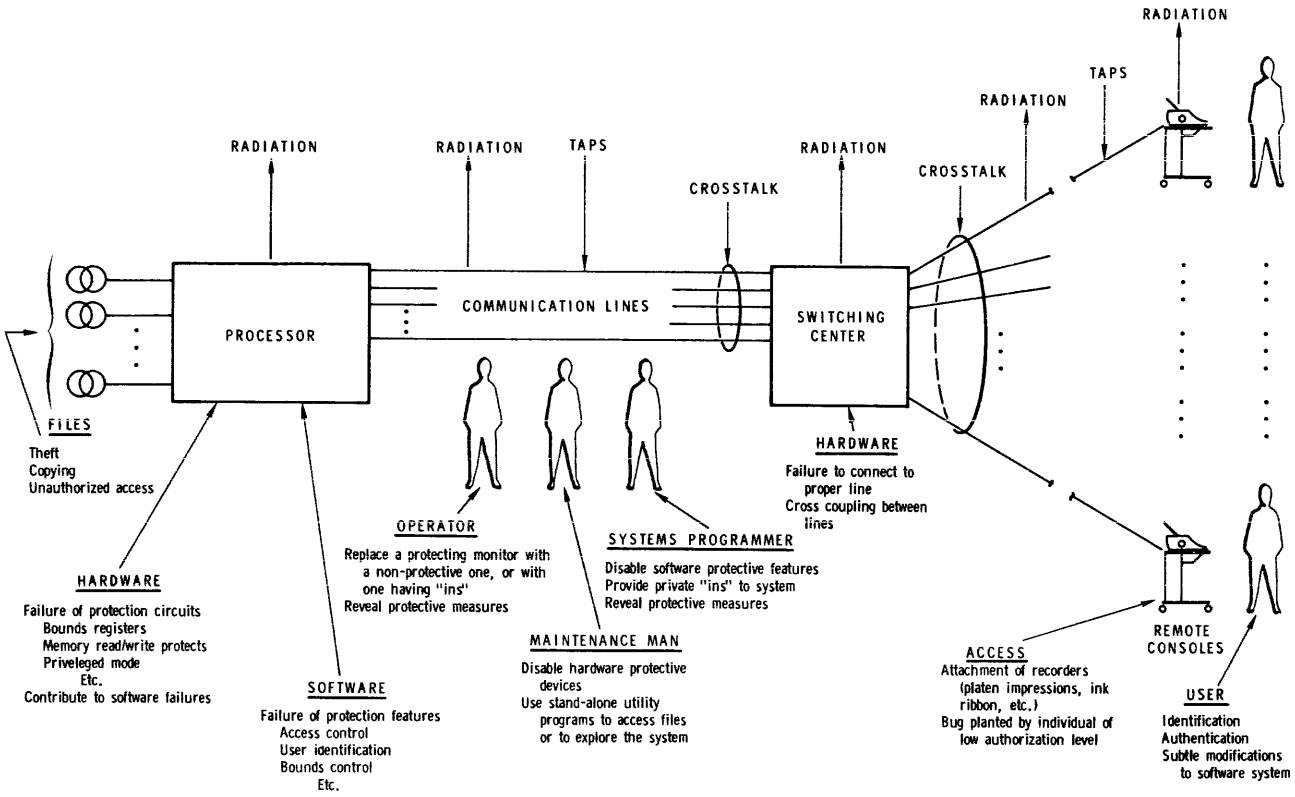


Figure 1 – Typical configuration of resource-sharing computer system

will refer to military or defense information or situations; "private" or "privacy," to the corresponding industrial, or non-military governmental situations. In each case, the individual authorized to receive the information will have "need to know" or "access authorization."

We will do the following in this session. In order to bring all of us to a common level of perspective on resource-sharing computer systems, I will briefly review the configuration of such systems and identify the major vulnerabilities to penetration and to leakage of information. The following paper by Mr. Peters will describe the security safeguards provided for a multi-programmed remote-access computer system. Then I will contrast the security and privacy situations, identifying similarities and differences. The final paper by Dr. Petersen and Dr. Turn will discuss technical aspects of security and privacy safeguards. Finally, we have a panel of three individuals who have faced the privacy problem in real-life systems; each will describe his views toward the problem, and his approach to a solution. In the end, it will fall upon each of you to conceive and implement satisfactory safeguards for the situation which concerns you.

A priori, we cannot be certain how dangerous a given vulnerability might be. Things which are serious for some computer systems may be only a nuisance for others. Let us take the point of view that we will not prejudge the risk associated with a given vulnerability or threat to privacy. Rather, let us try only to suggest some of the ways in which a computer system might divulge information to an unauthorized party in either the security or the privacy situation. We'll leave for discussion in the context of particular installations the question of how much protection we want to provide, what explicit safeguards must be provided, and how serious any particular vulnerability might be.

The hardware configuration of a typical resource-sharing computer system is shown in Figure 1. There is a central processor to which are attached computer-based files and a communication network for linking to remote users via a switching center. We observe first of all that the files may contain information of different levels of sensitivity or military classification; therefore, access to these files by users must be controlled. Improper or unauthorized access to a file can divulge information to the wrong person. Certainly, the file can also be stolen – a rather drastic divulgence of information. On the other hand, an unauthorized copy of a file might be made using the computer itself, and the copy revealed to unauthorized persons.

The central processor has both hardware and software components. So far as hardware is concerned, the circuits for such protections as bound registers, memory read-write protect, or privileged mode might fail and permit information to leak to improper destinations. A large variety of hardware failures might contribute to software failures which, in turn, lead to divulgence. Since the processor consists of high-speed electronic circuits, it can be expected that large quantities of electromagnetic energy will radiate; conceivably an eavesdropping third party might acquire sensitive information. Failure of the software may disable such protection features as access control, user identification, or memory bounds control, leading to improper routing of information.

Intimately involved with the central computer are three types of personnel: operators, programmers, and maintenance engineers. The operator who is responsible for minute-by-minute functioning of the system might reveal information by doing such things as replacing the correct monitor with a non-protecting one of his own, or perhaps with a rigged monitor which has special "ins" for unauthorized parties. Also, he might reveal to unauthorized parties some of the protective measures which are designed into the system. A co-operative effort between a clever programmer and an engineer could "bug" a machine for their own gain in such a sophisticated manner that it might remain unnoticed for an extended period. ("Bug" as just used does not refer to an error in a program, but to some computer equivalent of the famous transmitter in a martini olive.) Bugging of a machine could very easily appear innocent and open.

Operator-less machine systems are practical, and in principle one might conjecture that a machine could be bugged by an apparently casual passerby. There are subtle risks associated with the maintenance process. While attempting to diagnose a system failure, information could easily be generated which would reveal to the maintenance man how the software protections are coded. From that point, it might be easy to rewire the machine so that certain instructions appeared to behave normally, whereas in fact, the protective mechanisms could be bypassed.

While some of the things that I've just proposed require deliberate acts, others could happen by accident.

Thus, so far as the computing central itself is concerned, we have potential vulnerabilities in control of access to files; in radiation from the hardware; in hardware, software, or combined hardware-software failures; and in deliberate acts of penetration or accidental mistakes by the system personnel.

The communication links from the central processor to the switching center, and from the switching center to the remote consoles are similarly vulnerable. Any of the usual wiretapping methods might be employed to steal information from the lines. Since some communications will involve relatively high-frequency signals, electromagnetic radiation might be intercepted by an eavesdropper. Also, crosstalk between communication links might possibly reveal information to unauthorized individuals. Furthermore, the switching central itself might have a radiation or crosstalk vulnerability; it might fail to make the right connection and so link the machine to an incorrect user.

A remote console might also have a radiation vulnerability. Moreover, there is the possibility that recording devices of various kinds might be attached to the console to pirate information. Consideration might have to be given to destroying the ribbon in the printing mechanism, or designing the platen so that impressions could not be read from it.

Finally, there is the user of the system. Since his link to the computer is via a switching center, the central processor must make certain with whom it is conversing. Thus, there must be means for properly identifying the user; and this means must be proof against recording devices, pirating, unauthorized use, etc. Even after a user has satisfactorily established his identity, there remains the problem of verifying his right to have access to certain files, and possibly to certain components of the configuration. There must be a means for authenticating the requests which he will make of the system, and this means must be proof against bugging, recorders, pirating, unauthorized usage, etc. Finally, there is the ingenious user who skillfully invades the software system sufficiently to ascertain its structure, and to make changes which are not apparent to the operators or to the systems programmers, but which give him "ins" to normally unavailable information.

To summarize, there are human vulnerabilities throughout; individual acts can accidentally or deliberately jeopardize the protection of information in a system. Hardware vulnerabilities are shared among the computer, the communications system, and the consoles. There are software vulnerabilities; and vulnerabilities in the system's organization, e.g., access control, user identification and authentication. How serious any one of these might be depends on the sensitivity of the information being handled, the class of users, the operating environment, and certainly on the skill with which the network has been designed. In the most restrictive case, the network might have to be protected against all the types of

invasions which have been suggested plus many readily conceivable.

This discussion, although not an exhaustive consideration of all the ways in which a resource-sharing computer system might be either accidentally or deliberately penetrated for the purposes of unauthorized acquisition of information, has attempted to outline some of the major vulnerabilities which exist in modern computing systems. Succeeding papers in this session will address themselves to a more detailed examination of these vulnerabilities and to a discussion of possible solutions.

Security considerations in a multi-programmed computer system

by BERNARD PETERS
National Security Agency
Fort Meade, Maryland

INTRODUCTION

Security can not be attained in the absolute sense. Every security system seeks to attain a probability of loss which is commensurate with the value returned by the operation being secured. For each activity which exposes private, valuable, or classified information to possible loss, it is necessary that reasonable steps be taken to reduce the probability of loss. Further, any loss which might occur must be detected. There are several minimum requirements to establish an adequate security level for the software of a large multi-programmed system with remote terminals.

The management must be aware of the need for security safeguards. It must fully understand and be willing to support the cost of obtaining this security protection. It must be technically competent to judge whether or not a desired security level has been reached. It is important that the management understand that no software system can approach a zero risk of loss. It is necessary to reach an acceptable degree of risk.

As any professional in the field of computers knows, whether or not managment is sufficiently aware of and willing to pay the cost of a given software feature is not easy to determine. The costs of security are not exorbitant provided security is needed. However, they are not trivial or negligible. It is very important that the management have an understanding of how the security is gained and what administrative steps must be taken to maintain and protect the security level which has been gained. This paper will not fully explore certain problems which affect security as, for example, physical security which is protection of a space against penetration, or communications security or personnel security.

To obtain the security which software can contribute, the following principles must be followed:

1. The computer must operate under a monitor approved by appropriate authority. This can provide the authority for the expenditures which security may require. The question of appropriate authority is one not easily answered. For the individual business concern it obviously is corporate management. For contractors to the Armed Services it is a security officer of the service. Who can tell who is in charge in a University?

The computer must operate under a monitor because the monitor acts as the overall guard to the system. It provides protection against the operators and the users at the remote terminals. The monitor has a set of rules by which it judges all requested actions. It obeys only those requests for action which conform to the security principles necessary for the particular operation.

The discussion of what makes up an appropriate monitor is rather involved. Therefore, a more complete discussion of the security aspects of a monitor will be given after a summary of the overall security principles.

2. The computer must have adequate memory protect and privileged instructions. These are needed to limit user programs which must be considered to be hostile. Memory protect must be sufficient so that any reference, read or write, outside of the area assigned to a given user program must be detected and stopped. There are several forms of memory protect on the market which guard against out-of-bounds write, thus protecting program integrity, but they do not guard against illegal read. Read protect is as important as write protect, from a security standpoint, if classified material is involved.

The privileged instruction set must contain not only all Input/Output (I/O) commands but also every

command which could change a memory bound or protection barrier. To have less would be to reduce the whole system to a mockery. There can be no exception to the restriction that only the monitor can operate these privileged instructions.

3. The computer must have appropriate physical security protection to prevent local override of the monitor. It includes the obvious requirement that the computer can not be mounted in a subway car, subject to public tampering. The computer must be in an appropriately secure environment and not subject to intrusion by random individuals. But this is not sufficient. Certain key switches, such as those which affect the continued running of the equipment and the maintenance panel, must have simple physical barriers to prevent undetected intrusion from occurring. The senior operator provides protection against local override, but he cannot be expected to watch every switch continuously. Simple key-locks provide protection to supplement his attention span.

4. Electrical separation of peripheral devices is not necessary, provided the monitor has been approved by an appropriate authority. This reduces the probability of failure by eliminating troublesome separation hardware. It has been proposed by people who are first inquiring into security, that the peripheral devices be separated from the computer. A properly designed monitor for a multi-programming system will provide sufficient logical separation of peripheral devices to make electrical separation unnecessary. If the monitor is insecure and beyond securing, the system should not be used with the expectation that security will be provided.

5. The computer may operate in a multi-programmed or a multi-processor mode provided the monitor has been approved for use under such modes. In fact, to provide proper security it seems that the computer must be designed for the multi-programmed mode. In the rare case, when only one program is running, it is multi-programming with a single program. Multi-processors, for the purposes of this paper, are assumed to be a variant of multi-programming. It is obvious that this isn't true; however, it is felt that simple, logical extensions will carry the multi-programming philosophy into the multi-processor mode and still provide adequate protection.

6. Operating personnel must be cleared to appropriate levels. Trustworthy personnel must be available to enforce the rules against intrusion at the computer. This does not mean that every operator in the installation need be cleared, but it does mean that on every shift there must be at least one individual who is fully aware of the requirements of the operating doctrine which is needed to protect the secure data. He must completely understand what it takes to utilize the system appropriately and securely in any circumstance. All other operators must understand that there exists a protection philosophy. They must be subject to some form of discipline, such as termination or suspension for failure to obey regulations. On the other hand, the operating doctrine must be made clear and precise so that it can be obeyed. Specification of the operating doctrine is a nontrivial problem and can be very difficult depending upon the complexity of the tasks assigned to the operator. In the interest of keeping the operating doctrine under control it is proposed that the operator be designed out of the operation as much as possible.

7. A log of all significant events should be maintained both by the computer and the operating personnel. The form and contents of these logs should be approved by appropriate authority. The log is the ultimate defense against penetration. The question of whether or not some data have gone astray must be answered by the log. The log ascertains who does what, and to what extent he has attempted to do something illegal. It also records the utilization of files. It indicates whether or not some security restriction is significantly interferring with the production of the system. The log will provide the review and feed-back which will make it possible to relax security where it is overly stringent and it will provide the factual basis for increasing security where it has failed to prove a sufficient level of confidence.

8. Every user should be subject to common discipline and authority. He shall know and understand the conventions which are required of him to support the security system; it is particularly important that everyone who has access to the more highly classified levels of material fully understand his part in the protection of the data and the system. The remote terminals must also be electrically identithe station by its permanent connection to a specific set of hubs in the machine. Station identification can not be dependent upon the action of the user unless this action has been carefully considered for its security connotation.

9. It is possible that design considerations for the system require that an individual remote terminal change its particular security level upward at one time or another. It is necessary that this change be accomplished in a secure fashion. An acceptable method would be to issue the user of such a remote terminal a one-time list of five letter groups. The

user would be required to use each group, in turn, once and only once. This would make observing of the particular procedure for raising the station security level valueless to an observer. It has the further effect that should an observer obtain and use the next available five letter group, such use would cause a conflict with the legitimate user's call. Thus there would be detection of illegal use.

Let us now return to a discussion of the attributes of an acceptable monitor. The monitor is the key defense, the key security element in the system. It must be properly designed.

The cost of a properly designed monitor is probably not more than 10% greater than that of a monitor which is minimally acceptable for multi-programming. Monitors for multi-programming need a high degree of program and file integrity to be effective. Improvement of such a monitor to the point where it is acceptable for handling material in a secure fashion, is an achievable goal.

The monitor must perform all input/output (I/O) without exception. No user program can be permitted to utilize any I/O device, except via a call to the monitor. It is very important that the monitor control all I/O which could provide large amounts of information to a penetrating program. The monitor must manage the system clocks and the main console. If the maintenance panel is beyond management by the monitor, it must be secured.

The monitor must be carefully designed to limit the amount of critical coding. When an interrupt occurs, control is transferred to the monitor and the core is unprotected. The monitor must, as soon as reasonable, adjust the memory bounds to provide limits on even the monitor's own coding. This requires that the coding which receives interrupts be certified as error free for *all* possible inputs and timing. The greater portion of the coding of the monitor need not be certified error free to the same degree because it is operated under the same restraints as a user program. By keeping the amount of coding that can reference any part of core without restriction to a few well-tested units, confidence in the monitor can be established.

The monitor must be adequately tested. It is certainly necessary to adequately demonstrate the security capability to the governing authority. In addition to passing its initial acceptance the monitor must also test itself continuously. For instance, the memory bounds protection can be expected to fail with some probability. Every user program which conforms with the security safeguards will be expected to run without violating the memory bounds protection and therefore will not test such a feature. It is necessary, therefore, that some special program, or some part of the monitor deliberately and periodically violate the memory bounds protection to verify that the bounds protection checker is, in fact, working. It is extremely important that tests for all significant safeguards built into a monitor be periodically and deliberately verified. The confidence which management has in the security safeguards will, in part, rest on the evidence furnished by such self-tests.

The monitor must keep the user programs bounded by memory protect while they are operating. The individual user program must be free to reference its assigned area of core, but nowhere else. All I/O action, and any out-of-area reference by user programs, must be via calls to the monitor. This will protect information concerning the security levels and authorized users, files, and outputs from access by the operating elements of the monitor, such as loaders and terminators to be treated as though they were individual user programs, thus reducing the amount of coding that is outside the safeguards.

Authority to reference random access peripherals must be established by the monitor and all references must be checked for validity and authority. It is a small cost in a random access I/O routine to determine if a supplied address is legal and within bounds.

What is the monitor to do if there is a violation? If there is a violation of the memory bounds or the use of a privileged instruction by a user program, the monitor must immediately suspend the offending program and must make log entries. It must also prohibit the further use of the offending program by the user submitting the violating program until specifically authorized by a supervisor.

The suspension of violating program requests must be thorough and complete. If the task has been divided into multiple concurrent operating activities, all such activities must be terminated. If the task has resulted in a chain of requests, all such requests must be removed from the queue. Violation of security rules by any activity must result in a complete abort of all parts of that request. This is necessary to prevent a user program from making multiple tries against the security system.

Security rules cannot be suspended for debugging or program testing. It is clear that security rules cannot be suspended for debugging programs because the new program is the one most likely to violate security. The debugging system must live with the security restrictions although some concessions can be made for debugging. For example, one can flag a program that is in a debugging state. Then if a bounds violation occurs, the system could merely log it and send a dump of the program to the user rather than sounding a major alarm.

Tests of security flags must be fail safe. If a flag, when it referenced, is inconsistent or ambiguous it must fail to qualify. Thus security flags must be adequate. As an example, one could use a single bit, but then an error could change this bit to indicate a different but still valid combination; CLASSIFIED could too easily become UNCLASSIFIED. Therefore, single bits are not acceptable and the question then becomes "How many bits should one utilize?"

The specific configuration of bits is dependent upon the goals and requirements of the particular security system and the machine or bit handling capability of the individual machine. The author has used a 60 bit flag in a machine which has 30 bit words with half word capability. This led to four 15 bit configurations which were complementary to give verification. Forty bits would have been sufficient for the particular application, so the next higher multiple of word size was chosen.

Software security is derived by the proper application of a few sound principles which must be consistently applied in many small actions. With a proper understanding and careful review, software security can be maintained so long as physical and personnel security are assumed. The principles set forth in this paper have been generalized from the specific development of a specific system which dealt with multi-levels of classified information.

Security and privacy: similarities and differences

by WILLIS H. WARE
The RAND Corporation
Santa Monica, California

For the purposes of this paper we will use the term "security" when speaking about computer systems which handle classified defense information, and "privacy" in regard to those computer systems which handle non-defense information which nonetheless must be protected because it is in some respect sensitive. It should be noted at the outset that the context in which security must be considered is quite different from that which can be applied to the privacy question. With respect to classified military information there are federal regulations which establish authority, and discipline to govern the conduct of people who work with such information. Moreover, there is an established set of categories into which information is classified. Once information is classified Confidential, Secret, or Top Secret, there are well-defined requirements for its protection, for controlling access to it, and for transmitting it from place to place. In the privacy situation, analogous/conditions may exist only in part or not at all.

There are indeed Federal and State statutes which protect the so-called "secrecy of communication." But it remains to be established that these laws can be extended to cover or interpreted as applicable to the unauthorized acquisition of information from computer equipment. There are also laws against thievery; and at least one case involving a programmer and theft of privileged information has been tried. The telephone companies have formulated regulations governing the conduct of employees (who are subject to "secrecy of communication" laws) who may intrude on the privacy of individuals; perhaps this experience can be drawn upon by the computer field.

Though there apparently exist fragments of law and some precedents bearing on the protection of information, nonetheless the privacy situation is not so neatly circumscribed and tidy as the security situation. Privacy simply is not so tightly controlled. Within computer networks serving many companies, organizations, or agencies, there may be no uniform governing authority; an incomplete legal framework; no established discipline, or perhaps not even a code of ethics among users. At present there is not even a commonly accepted set of categories to describe levels of sensitivity for private information.

Great quantities of private information are being accumulated in computer files; and the incentives to penetrate the safeguards to privacy are bound to increase. Existing laws may prove inadequate, or may need more vigorous enforcement. There may be need for a monitoring and enforcement establishment analogous to that in the security situation. In any event, it can not be taken for granted that there now exist adequate legal and ethical umbrellas for the protection of private information.

The privacy problem is really a spectrum of problems. At one end, it may be necessary to provide only a very low level of protection to the information for only a very short time; at the opposite end, it may be necessary to invoke the most sophisticated techniques to guarantee protection of information for extended periods of time. Federal regulations state explicitly what aspect of national defense will be compromised by unauthorized divulgence of each category of classified information. There is no corresponding particularization of the privacy situation; the potential damage from revealing private information is nowhere described in such absolute terms. It may be that a small volume of information leaked from a private file may involve inconsequential risk. For example, the individual names of a company's employees is probably not even sensitive, whereas the complete file of employees could well be restricted. Certainly the "big brother" spectre raised by recent Congressional hearings on "invasion of privacy" via massive computer files is strongly related to the volume of information at risk.

Because of the diverse spread in the privacy situation, the appearance of the problem may be quite different from its reality. One would argue on principle that maximum protection should be given to all information labeled private; but if privacy of information is not protected by law and authority, we can expect that the owner of sensitive information will require a system designed to guarantee protection only against the threat as he sees it. Thus, while we might imagine very sophisticated attacks against private files, the reality of the situation may be that much simpler levels of protection will be accepted by the owners of the information.

In the end, an engineering trade-off question must be assessed. The value of private information to an outsider will determine the resources he is willing to expend to acquire it. In turn, the value of the information to its owner is related to what he is willing to pay to protect it. Perhaps this game-like situation can be played out to arrive at a rational basis for establishing the level of protection. Perhaps a company or governmental agency—or a group of companies or agencies, or the operating agent of a multi-access computer service—will have to establish its own set of regulations for handling private information. Further, a company or agency may have to establish penalties for infractions of these regulations, and perhaps even provide extra remuneration for those assuming the extraordinary responsibility of protecting private information.

The security measures deemed necessary for a multi-processing remote terminal computer system operating in a military classified environment have been discussed in the volume.* This paper will compare the security situation with the privacy situation, and suggest issues to be considered when designing a computer system for guarding private information. Technology which can be applied against the design problem is described elsewhere.†

First of all, note that the privacy problem is to some extent present whenever and wherever sharing of the structures of a computer system takes place. A time-sharing system slices time in such a way that each user gets a small amount of attention on some periodic basis. More than one user program is resident in the central storage at one time; and hence, there are obvious opportunities for leakage of information from one program to another, although the problem is alleviated to some extent in systems operating in an interpretive software mode. In a multi-programmed computer system it is also true that more than one user program is normally resident in the core store at a time. Usually, a given program is not executed without interruption; it must share the central storage and perhaps other levels of storage with other programs. Even in the traditional batch-operated system there can be a privacy problem. Although only one program is usually resident in storage at a time, parts of other programs reside on magnetic tape or discs; in principle, the currently executing program might accidentally reference others, or cause parts of previous programs contained on partially re-used magnetic tape to be outputed.

Thus, unless a computer system is completely stripped of other programs—and this means clearing or removing access to all levels of storage—privacy infractions are possible and might permit divulgence of information from one program to another.

Let us now reconsider the points raised in the Peters* paper and extend the discussion to include the privacy situation.

(1) The problem of controlling user access to the resource-sharing computer system is similar in both the security and privacy situations. It has been suggested that one-time passwords are necessary to satisfactorily identify and authenticate the user in the security situation. In some university time-sharing systems, permanently assigned passwords are considered acceptable for user identification. Even though printing of a password at the console can be suppressed, it is easy to ascertain such a password by covert means; hence, repeatedly used passwords may prove unwise for the privacy situation.

(2) The incentive to penetrate the system is present in both the security and privacy circumstances. Revelation of military information can degrade the country's defense capabilities. Likewise, divulgence of sensitive information can to some extent damage other parties or organizations. Private information will always have some value to an outside party, and it must be expected that penetrations will be attempted against computer systems handling such information. It is conceivable that the legal liability for unauthorized leaking of sensitive information may become as severe as for divulging classified material.

(3) The computer hardware requirements appear to be the same for the privacy and security situations. Such features as memory read-write protection, bounds registers, privileged instructions, and a privileged mode of operation are required to protect

*Peters, B., "Security Considerations in a Multi-Programmed System".

†Petersen, H. E., and R. Turn, Systems Implications of Privacy."

*Peters, B., *loc cit.*

information, be it classified or sensitive. Also, overall software requirements seem similar, although certain details may differ in the privacy situation because of communication matters or difference in user discipline.

(4) The file access and protection problem is similar under both circumstances. Not all users of a shared computer-private system will be authorized access to all files in the system, just as not all users of a secure computer system will be authorized access to all files. Hence, there must be some combination of hardware and software features which controls access to the on-line classified files in conformance with security levels and need-to-know restrictions and in conformance with corresponding attributes in the privacy situation. As mentioned earlier, there may be a minor difference relative to volume. In classified files, denial of access must be absolute, whereas in private files access to a small quantity of sensitive information might be an acceptable risk.

(5) The philosophy of the overall system organization will probably have to be different in the privacy situation. In the classified defense environment, users are indoctrinated in security measures and their personal responsibility can be considered as part of the system design. Just as the individual who finds a classified document in a hallway is expected to return it, so the man who accidentally receives classified information at his console is expected to report it. The users in a classified system are subject to the regulations, authority, and discipline of a governmental agency. Similar restrictions may not prevail in a commercial or industrial resource-sharing computer network, nor in government agencies that do not operate within the framework of government classification. In general, it would appear that one cannot exploit the good will of users as part of a privacy system's design. On the other hand, the co-operation of users may be part of the design philosophy if it proves possible to impose a uniform code of ethics, authority, and discipline within a multi-access system. Uniform rules of behavior might be possible if all users are members of the same organization, but quite difficult or impossible if the users are from many companies or agencies.

(6) The certifying authority is certainly different in the two situations. It is easy to demonstrate that the total number of internal states of a computer is so enormous that some of them will never prevail in the lifetime of the machine. It is equally easy to demonstrate that large computer programs have a huge number of internal paths, which implies the potential existence of error conditions which may appear rarely or even only once. Monitor programs governing the internal scheduling and operation of multi-programmed, time-sharing or batch-operated machines are likely to be extensive and complex; and if security or privacy is to be guaranteed, some authority must certify that the monitor is properly programmed and checked out. Similarly, the hardware must also be certified to possess appropriate protective devices.

In a security situation, a security officer is responsible for establishing and implementing measures for the control of classified information. Granted that he may have to take the word of computer experts or become a computer expert himself, and granted that of itself his presence does not solve the computer security problem, there is nonetheless at least an assigned, identifiable responsible authority. In the case of the commercial or industrial system, who is the authority? Must the businessman take the word of the computer manufacturer who supplied the software? If so, how does he assure himself that the manufacturer hasn't provided "ins" to the system that only he, the manufacturer, knows about? Must the businessman create his own analog of defense security practices?

(7) Privacy and security situations are certainly similar in that deliberate penetrations must be anticipated, if not expected; but industrial espionage against computers may be less serious. On the other hand, industrial penetrations against computers could be very profitable and perhaps safer from a legal viewpoint.

It would probably be difficult for a potential penetrator to mount the magnitude of effort against an industrial resource-sharing computer system that foreign agents are presumed to mount against secrecy systems of other governments. To protect against large-scale efforts, an industry-established agency could keep track of major computing installations and know where penetration efforts requiring heavy computer support might originate. On the other hand, the resourceful and insightful individual can be as great a threat to the privacy of a system. If one can estimate the nature and extent of the penetration effort expected against an industrial system, perhaps it can be used as a design parameter to establish the level of protection for sensitive information.

(8) The security and privacy situations are certainly similar in that each demands secure communication circuits. For the most part, methods for assuring the security of communication channels have been the exclusive domain of the military and government. What about the non-government user? Could the specifications levied on common carriers in their

implied warranty of a private circuit be extended? Does the problem become one for the common carriers? Must they develop communication security equipment? If the problem is left to the users, does each do as he pleases? Might it be feasible to use the central computer itself to encode information prior to transmission? If so, the console will require special equipment for decoding the messages.

(9) Levels of protection for communications are possibly different in the two situations. If one believes that a massive effort at penetration could not be mounted against a commercial private network, a relatively low-quality protection for communication would be sufficient. On the other hand, computer networks will inevitably go international. Then what? A foreign industry might find it advantageous to tap the traffic of U.S. companies operating an international and presumably private computer network. Might it be that for reasons of national interest we will someday find the professional cryptoanalytic effort of a foreign government focused on the privacy-protecting measures of a computer network?

If control of international trade were to become an important instrument of government policy, then any international communications network involved with industrial or commercial computer-private systems will need the best protection that can be provided.

This paper has attempted to identify and briefly discuss the differences and similarities between computer systems operating with classified military information and computer systems handling private or sensitive information. Similar hardware and software and systems precautions must be taken. In most respects, the differences between the two situations are only of degree. However, there are a few aspects in which the two situations genuinely differ in kind, and on these points designers of a system must take special note. The essential differences between the two situations appear to be the following:

(1) Legal foundations for protecting classified information are well established, whereas in the privacy situation a uniform authority over users and a penalty structure for infractions are lacking. We may not be able to count on the good will and disciplined behavior of users as part of the protective measures.

(2) While penetrations can be expected against both classified and sensitive information, the worth of the material at risk in the two situations can be quite different, not only to the owner of the data but also to other parties and to society.

(3) The magnitude of the resources available for protection and for penetration are markedly smaller in the privacy situation.

(4) While secure communications are required in both situations, there are significant differences in details. In the defense environment, protected communications are the responsibility of a government agency, appropriate equipment is available, and the importance of protection over-rides economic considerations. In the privacy circumstance, secure satisfactory communication equipment is generally not available, and the economics of protecting communications is likely to be more carefully assessed.

(5) Some software details have to be handled differently in the privacy situation to accommodate differences in the security of communications.

It must be remembered that since the Federal authority and regulations for handling classified military information do not function for private or sensitive information, it does not automatically follow that a computer network designed to safely protect classified information will equally well protect sensitive information. The all important difference is that the users of a computer-private network may not be subject to a common authority and discipline. But even if they are, the strength of the authority may not be adequate to deter deliberate attempts at penetration.

System implications of information privacy

by H. E. PETERSEN and R. TURN
The RAND Corporation
Santa Monica, California

INTRODUCTION

Recent advances in computer time-sharing technology promise information systems which will permit simultaneous on-line access to many users at remotely located terminals. In such systems, the question naturally arises of protecting one user's stored programs and data against unauthorized access by others. Considerable work has already been done in providing protection against accidental access due to hardware malfunctions or undebugged programs. Protection against deliberate attempts to gain access to private information, although recently discussed from a philosophical point of view,[1-4] has attracted only fragmentary technical attention. This paper presents a discussion of the threat to information privacy in non-military information systems, applicable countermeasures, and system implications of providing privacy protection.

The discussion is based on the following model of an information system: a central processing facility of one or more processors and an associated memory hierarchy; a set of information files—some private, others shared by a number of users; a set of public or private query terminals at geographically remote locations; and a communication network of common carrier, leased, or private lines. This time-shared, on-line system is referred to throughout as "the system."

Threats to information privacy

Privacy of information in the system is lost either by accident or deliberately induced disclosure. The most common causes of *accidental* disclosures are failures of hardware and use of partially debugged programs. Improvements in hardware reliability and various memory protection schemes have been suggested as countermeasures. *Deliberate* efforts to infiltrate an on-line, time-shared system can be classified as either passive or active.

Passive infiltration may be accomplished by wiretapping or by electromagnetic pickup of the traffic at any point in the system. Although considerable effort has been applied to counter such threats to defense communications, nongovernmental approaches to information privacy usually assume that communication lines are secure, when in fact *they are the most vulnerable part of the system.* Techniques for penetrating communication networks may be borrowed from the well-developed art of listening in on voice conversations.[5,6] (While the minimum investment is equipment is higher than that required to obtain a pair of headsets and a capacitor, it is still very low since a one-hundred-dollar tape recorder and a code conversion table suffice.) Clearly, digital transmission of information does not provide any more privacy than, for example, Morse code. Nevertheless, some users seem willing to entrust to digital systems valuable information that they would not communicate over a telephone.

Active infiltration—an attempt to enter the system to directly obtain or alter information in the files—can be overtly accomplished through normal access procedures by:

- Using legitimate access to a part of the system to ask unauthorized questions (e.g., requesting payroll information or trying to associate an individual with certain data), or to "browse" in unauthorized files.
- "Masquerading" as a legitimate user after having obtained proper identifications through wiretapping or other means.
- Having access to the system by virtue of a position with the information center or the communication network but without a "need to know" (e.g., system programmer, operator, maintenance, and management personnel).

Or an active infiltrator may attempt to enter the system covertly i.e., avoiding the control and protection programs) by:

- Using entry points planted in the system by unscrupulous programmers or maintenance engineers, or probing for and discovering "trap doors" which may exist by virtue of the combinatorial aspects of the many system control variables (similar to the search for "new and useful" operation codes—a favorite pastime of machine-language programmers of the early digital computers).
- Employing special terminals tapped into communication channels to effect:
  —"piggy back" entry into the system by selective interception of communications between a user and the processor, and then releasing these with modifications or substituting entirely new messages while returning an "error" message;
  —"between lines" entry to the system when a legitimate user is inactive but still holds the communication channel;
  —cancellation of the user's sign-off signals, so as to continue operating in his name.
  In all of these variations the legitimate user provides procedures for obtaining proper access. The infiltrator is limited, however, to the legitimate user's authorized files.

More than an inexpensive tape recorder is required for active infiltration, since an appropriate terminal and entry into the communication link are essential. In fact, considerable equipment and know-how are required to launch sophisticated infiltration attempts.

The objectives of infiltration attempts against information systems have been discussed by a number of authors[1-3,7] from the point of view of potential payoff. We will merely indicate the types of activities that an infiltrator may wish to undertake:

- Gaining access to desired information in the files, or discovering the information interests of a particular user.
- Changing information in the files (including destruction of entire files).
- Obtaining free computing time or use of proprietary programs.

Depending on the nature of the filed information, a penetration attempt may cause no more damage than satisfying the curiosity of a potentially larcenous programmer. Or it may cause great damage and result in great payoffs; e.g., illicit "change your dossier for a fee," or industrial espionage activities. (See Table I for a summary of threats to information privacy.)

More sophisticated infiltration scenarios can be conceived as the stakes of penetration increase. The threat to information privacy should not be taken lightly or brushed aside by underestimating the resources and ingenuity of would-be infiltrators.

Countermeasures

The spectrum of threats discussed in the previous section can be countered by a number of techniques and procedures. Some of these were originally introduced into time-shared, multi-user systems to prevent users from inadvertently disturbing each other's programs,[8] and then expanded to protect against accidental or deliberately induced disclosures

TABLE I. Summary of Threats to Information Privacy

Nature of Infiltration	Means	Effects
Accidental	Computer malfunctioning; user errors; undebugged programs	Privileged information dumped at wrong terminals, printouts, etc.
Deliberate Passive	Wiretapping, electromagnetic pickup, examining carbon papers, etc.	User's interest in information revealed; content of communications revealed
Deliberate Active	Entering files by: "Browsing"; "Masquerading"; "Between lines"; "Piggy-back" penetration	Specific information revealed or modified as a result of infiltrator's actions

of information.[8,9] Others found their beginning in requirements to protect privacy in communication networks.[10] In the following discussion, we have organized the various countermeasures into several classes.

Access management

These techniques are aimed at preventing unauthorized users from obtaining services from the system or gaining access to its files. The procedures involved are authorization, identification, and authentication. *Authorization* is given for certain users to enter the system, gain access to certain files, and request certain types of information. For example, a researcher may be permitted to compile earnings statistics from payroll files but not to associate names with salaries. Any user attempting to enter the system must first *identify* himself and his location (i.e., the remote terminal he is using), and then *authenticate* his identification. The latter is essential if information files with limited access are requested; and is desirable to avoid mis-charging of the computing costs. The identification-authentication steps may be repeated any number of times (e.g., when particularly sensitive files are requested).

Requirements for authentication may also arise in reverse, i.e., the processor identifies and authenticates itself to the user with suitable techniques. This may satisfy the user that he is not merely conversing with an infiltrator's console. Applied to certain messages from the processor to the user (e.g., error messages or requests for repetition) these could be authenticated as coming from the processor and not from a piggy-back penetrator. (Since various specific procedures are applied to different parts of the system, a detailed discussion of their structures and effectiveness is presented in Sec. IV.)

Processing restrictions

Although access control procedures can eliminate the simple threats from external sources, they cannot stop sophisticated efforts nor completely counter legitimate users or system personnel inclined to browse. An infiltrator, once in the system will attempt to extract, alter, or destroy information in the files. Therefore, some processing restrictions (in addition to the normal memory protection features) need to be inposed on files of sensitive information. For example, certain removable files may be mounted on drives with disabled circuits, and alterations of data performed only after requests are authenticated by the controller of each file. Copying complete files (or large parts of files) is another activity where processing control need to be imposed—again in the form of authentication of file controllers.

In systems where very sensitive information is handled, processing restrictions could be imposed on specific users in instances of "suspicious" behavior. For example, total cancellation of any program attempting to enter unauthorized files may be an effective countermeasure against browsing.[9]

Threat monitoring

Threat monitoring concerns detection of attempted or actual penetrations of the system or files either to provide a real-time response (e.g., invoking job cancellation, or starting tracing procedures) or to permit *post facto* analysis. Threat monitoring may include recording of all rejected attempts to enter the system or specific files, use of illegal access procedures, unusual activity involving a certain file, attempts to write into protected files, attempts to perform restricted operations such as copying files, excessively long periods of use, etc. Periodic reports to users on file activity may reveal possible misuse or tampering, and prompt stepped-up auditing along with a possible real-time response. Such reports may range from a page by page synopsis of activity during the user session, to a monthly analysis and summary.

Privacy transformations

Privacy transformations[10,11] are techniques for coding the data in user-processor communications or in files to conceal information. They could be directed against passive (e.g., wiretapping) as well as sophisticated active threats (e.g., sharing a user's identification and communication link by a "piggyback" infiltrator), and also afford protection to data in removable files against unauthorized access or physical loss.

A privacy transformation consists of a set of reversible logical operations on the individual characters of an information record, or on sets of such records. Reversibility is required to permit recovery (decoding) of the original information from the encoded form. Classes of privacy transformations include:

- *Substitution*—replacement of message characters with characters or groups of characters in the same or a different alphabet in a one-to-one manner (e.g., replacing alphanumeric characters with groups of binary numerals).
- *Transposition*—rearrangement of the ordering of characters in a message.
- *Addition*—using appropriate "algebra" to combine characters in the message with encoding sequences of characters (the "key") supplied by the user.

Well-known among a number of privacy transfor-

mations of the "additive" type[11] are the "Vigenere cipher," where a short sequence of characters are repeatedly used to combine with the characters of the message, and the "Vernam system," where the user-provided sequence is at least as long as the message. Successive applications of several transformations may be used to increase the complexity.

In general, the user of a particular type of privacy transformation (say, the substitution of characters in the same alphabet has a very large number of choices of transformations in that class (e.g., there are 26 factorial -4×10^{26} – possible substitution schemes of the 26 letters of the English alphabet). The identification of a particular transformation is the "key" chosen by the user for encoding the message.

Any infiltrator would naturally attempt to discover the key – if necessary, by analyzing an intercepted encoded message. The effort required measures the "work factor" of the privacy transformation, and indicates the amount of protection provided, assuming the key cannot be stolen, etc. The work factor depends on the type of privacy transformations used, the statistical characteristics of the message language, the size of the key space, etc. A word of caution against depending too much on the large key space: Shannon[11] points out that about 3×10^{12} years would be required, on the average, to discover the key used in the aforementioned substitution cipher with 26 letters by an exhaustive trial-and-error method (eliminating one possible key every microsecond). However, according to Shannon, repeatedly dividing the key space into two sets of roughly an equal number of keys and eliminating one set each trial (similar to coin-weighing problems) would require only 88 trials.

The level of work factor which is critical for a given information system depends, of course, on an estimate of the magnitude of threats and of the value of the information. For example, an estimated work factor of one day of continuous computation to break a single key may be an adequate deterrent against a low-level threat.

Other criteria which influence the selection of a class of privacy transformations are:[11]

- *Length of the key* – Keys require storage space, must be protected, have to be communicated to remote locations and entered into the system, and may even require memorization. Though generally a short key length seems desirable, better protection can be obtained by using a key as long as the message itself.
- *Size of the key space* – The number of different privacy transformations available should be as large as possible to discourage trial-and-error approaches, and to permit assignment of unique keys to large numbers of users and changing of keys at frequent intervals.
- *Complexity* – Affects the cost of implementation of the privacy system by requiring more hardware or processing time, but may also improve the work factor.
- *Error sensitivity* – The effect of transmission errors or processor malfunctioning may make decoding impossible.

Other criteria are, of course, the cost of implementation and processing time requirements which depend, in part, on whether the communication channel or the files of the system are involved (see Sec. IV).

Integrity management

Important in providing privacy to an information system is verification that the system software and hardware perform as specified – including an exhaustive initial verification of the programs and hardware, and later, periodic checks. Between checks, strict controls should be placed on modifications of software and hardware. For example, the latter may be kept in locked cabinets equipped with alarm devices. Verification of the hardware integrity after each modification or repair should be a standard procedure, and inspection to detect changes of the emissive characteristics performed periodically.

Integrity of the communication channel is a far more serious problem and, if common carrier connections are employed, it would be extremely difficult to guarantee absence of wiretaps.

Personnel integrity (the essential element of privacy protection) poses some fundamental questions which are outside the scope of this paper. The assumptions must be made, in the interest of realism, that not everyone can be trusted. System privacy should depend on the integrity of as few people as possible.

System aspects of information privacy

As pointed out previously, not all parts of an information system are equally vulnerable to threats to information privacy, and different countermeasures may be required in each part to counter the same level of threat. The structure and functions of the information processor, the files, and the communication network with terminals are, in particular, sufficiently different to warrant separate discussion of information privacy in these subsystems.

Communication lines and terminals

Since terminals and communication channels are the principal user-to-processor links, privacy of

information in this most vulnerable part of the system is essential.

Wiretapping: Many users spread over a wide area provide many opportunities for wiretapping. Since the cost of physically protected cables is prohibitive, there are no practical means available to prevent this form of entry. As a result, only through protective techniques applied at the terminals and at the processor can the range of threats from simple eavesdropping to sophisticated entry through special terminals be countered. While a properly designed password identification-authentication procedure is effective against some active threats, it does not provide any protection against the simplest threat—eavesdropping—nor against sophisticated "piggy-back" entry. The only broadly effective countermeasure is the use of privacy transformations.

Radiation: In addition to the spectrum of threats arising from wiretapping, electromagnetic radiation from terminals must be considered.[12]

Electromagnetic radiation characteristics will depend heavily on the type of terminal, and may in some cases pose serious shielding and electrical-filtering problems. More advanced terminals using cathode ray tubes for information display may create even greater problems in trying to prevent what has been called "tuning in the terminal on Channel 4."

Use of privacy transformations also helps to reduce some of the problems of controlling radiation. In fact, applying the transformation as close to the electromechanical converters of the terminal as possible minimizes the volume that must be protected, and reduces the extent of vulnerable radiation characteristics.

Obviously, the severity of these problems depends upon the physical security of the building or room in which the terminal is housed. Finally, proper handling and disposal of typewriter ribbons, carbon papers, etc., are essential.

Operating modes: Whether it would be economical to combine both private and public modes of operation into a single standard terminal is yet to be determined; but it appears desirable or even essential to permit a private terminal to operate in the public mode, although the possibility of compromising the privacy system must be considered. For example, one can easily bypass any special purpose privacy hardware by throwing a switch manually or by computer, but these controls may become vulnerable to tampering. The engineering of the terminal must, therefore, assure reasonable physical, logical, and electrical integrity for a broad range of users and their privacy requirements.

Terminal identification: An unambiguous and authenticated identification of a terminal is required for log-in, billing, and to permit system initiated call-back for restarting or for "hang-up and redial"[13] access control procedures. The need for authentication mainly arises when the terminal is connected to the processor via the common-carrier communication lines, where tracing of connections through switching centers is difficult. If directly wired connections are used, neither authentication nor identification may be required, since (excluding wiretaps) only one terminal can be on a line.

Identification of a terminal could involve transmission of an internally (to the terminal) generated or user-entered code word consisting, for example, of two parts: one containing a description or name of the terminal; the other, a password (more about these later) which authenticates that the particular terminal is indeed the one claimed in the first part of the code word. Another method suggested for authenticating the identity of a terminal is to use computer hang-up and call-back procedures.[13] After terminal identification has been satisfactorily established, the processor may consult tables to determine the privacy level of the terminal; i.e., the users admitted to the terminal, the protection techniques required, etc.

User identification: As with a terminal, identifying a user may require stating the user's name and account number, and then authenticating these with a password from a list, etc.

If the security of this identification process is adequate, the normal terminal input mechanisms may be used; otherwise, special features will be required. For example, hardware to accept and interpret coded cards might be employed, or sets of special dials or buttons provided. Procedures using the latter might consist of operating these devices in the correct sequence.

In some instances, if physical access to a terminal is appropriately controlled, terminal identification may be substituted for user identification (and vice versa).

Passwords: Clearly, a password authenticating a user or a terminal would not remain secure indefinitely. In fact, in an environment of potential wiretapping or radiative pickup, a password might be compromised by a single use. Employing lists of randomly selected passwords, in a "one-time-use" manner where a new word is taken from the list each time authentication is needed has been suggested as a countermeasure under such circumstances.[9] One copy of such a list would be stored in the processor, the other maintained in the terminal or carried by the user. After signing in, the user takes the next work on the list, transmits it to the processor and then crosses it off. The processor compares the received password with the next word in its own list and

permits access only when the two agree. Such password lists could be stored in the terminal on punched paper tape, generated internally by special circuits, or printed on a strip of paper. The latter could be kept in a secure housing with only a single password visible. A special key lock would be used to advance the list. Since this method of password storage precludes automatic reading, the password must be entered using an appropriate input mechanism.

The protection provided by use of once-only passwords during sign-in procedures only is not adequate against more sophisticated "between lines" entry by an infiltrator who has attached a terminal to the legitimate user's line. Here the infiltrator can use his terminal to enter the system between communications from the legitimate user. In this situation the use of once-only passwords must be extended to each message generated by the user. Automatic generation and inclusion of authenticating passwords by the terminal would now be essential for smoothness of operation; and lists in the processor may have to be replaced by program or hardware implemented password generators.

Privacy transformations: The identification procedures discussed above do not provide protection against passive threats through wire-tapping, or against sophisticated "piggy-back" entry into the communication link. An infiltrator using the latter technique would simply intercept messages—password and all—and alter these or insert his own (e.g., after sending the user an error indication). Authentication of each message, however, will only prevent a "piggy-back" infiltrator from using the system after cancelling the sign-off statement of the legitimate user.

Although it may be conceivable that directly wired connections could be secured against wiretapping, it would be nearly impossible to secure common-carrier circuits. Therefore, the use of privacy transformations may be the only effective countermeasures against wire-tapping and "piggy-back" entry, as they are designed to render encoded messages unintelligible to all but holders of the correct key. Discovering the key, therefore, is essential for an infiltrator. The effort required to do this by analyzing intercepted encoded messages (rather than by trying to steal or buy the key) is the "work factor" of a privacy transformation. It depends greatly on the type of privacy transformations used, as well as on the knowledge and ingenuity of the infiltrator.

The type of privacy transformation suitable for a particular communication network and terminals depends on the electrical nature of the communication links, restrictions on the character set, structure and vocabulary of the query language and data, and on the engineering aspects and cost of the terminals. For example, noisy communication links may rule out using "auto key" transformation[11] (e.g., those where the message itself is the key for encoding); and highly structured query languages may preclude direct substitution schemes, since their statistics would not be obscured by substitution and would permit easy discovery of the substitution rules. Effects of character-set restrictions on privacy transformations become evident where certain characters are reserved for control of the communication net, terminals, or the processor (e.g., "end of message," "carriage return," and other control characters). These characters should be sent in the clear and appear only in their proper locations in the message, hence imposing restrictions on the privacy transformation.

A number of other implications of using privacy transformations arise. For example, in the private mode of terminal operation the system may provide an end-to-end (terminal-to-processor) privacy transformation with a given work factor, independently of the user's privacy requirements. If this work factor is not acceptable to a user, he should be permitted to introduce a preliminary privacy transformation using his own key in order to increase the combined work factor. If this capability is to be provided at the terminal, there should be provisions for inserting additional privacy-transformation circuitry.

Another, possibly necessary feature, might allow the user to specify by appropriate statements whether privacy transforms are to be used or not. This would be part of a general set of "privacy instructions" provided in the information-system operating programs. Each change from private to public mode, especially when initiated from the terminal, should be authenticated.

Files

While the above privacy-protection techniques and access-control procedures for external terminals and the communication network may greatly reduce the threat of infiltration by those with no legitimate access, they do not protect information against (1) legitimate users attempting to browse in unauthorized files, (2) access by operating and maintenance personnel, or (3) physical acquisition of files by infiltrators.

A basic aspect of providing information privacy to files is the right of a user to total privacy of his files—*even the system manager of the information center should not have access.* Further, it should be possible to establish different levels of privacy in files. That is, it should be feasible to permit certain of a group of users to have access to all of the com-

pany's files, while allowing others limited access to only some of these files.

In this context certain standard file operations—such as file copying—would seem inappropriate, if permitted in an uncontrolled manner, since it would be easy to prepare a copy of a sensitive file and maintain it under one's own control for purposes other than authorized. Similarly, writing into files should be adequately controlled. For example, additions and deletions to certain files should be authorized only after proper authentication. It may even be desirable to mount some files on drives with physically disabled writing circuits.

Access control: Control of access to the files would be based on maintaining a list of authorized users for each file, where identification and authentication of identity (at the simplest), is established by the initial sign-in procedure. If additional protection is desired for a particular file, either another sign-in password or a specific file-access password is requested to reauthenticate the user's identity. The file-access passwords may be maintained in a separate list for each authorized user, or in a single list. If the latter, the system would ask the user for a password in a specific location in the list (e.g., the tenth password). Although a single list requires less storage and bookkeeping, it is inherently less secure. Protection of files is thus based on repeated use of the same requirements—identification and authentication—as for initial access to the system during sign-in. This protection may be inadequate, however, in systems where privacy transformations are not used in the communication net (i.e., "piggy-back" infiltration is still possible.)

Physical vulnerability: An additional threat arises from possible physical access to files. In particular, the usual practice of maintaining backup files (copies of critical files for recovery from drastic system failures) compounds this problem. Storage, transport, and preparation of these files all represent points of vulnerability for copying, theft, or an off-line print out. Clearly, possession of a reel of tape, for example, provides an interloper the opportunity to peruse the information at his leisure. Applicable countermeasures are careful storage and transport, maintaining the physical integrity of files throughout the system, and the use of privacy transformations.

Privacy transformations: As at the terminals and in the communication network, privacy transfer—mations could be used to protect files against failure of normal access control or physical protection procedures. However, the engineering of privacy transformations for files differs considerably:

- Both the activity level and record lengths are considerably greater in files;
- Many users, rather than one, may share a file;
- Errors in file operations are more amenable to detection and control than those in communication links, and the uncorrected error rates are lower;
- More processing capability is available for the files, hence more sophisticated privacy transformations can be used;
- Many of the files may be relatively permanent and sufficiently large, so that frequent changes of keys are impractical due to the large amount of processing required. Hence, transformations with higher work factors could be used and keys changed less frequently.

It follows that the type of privacy transformation adequate for user-processor communications may be entirely unacceptable for the protection of files.

The choice of privacy transformations for an information file depends heavily on the amount of file activity in response to a typical information request, size and structure of the file (e.g., short records, many entry points), structure of the data within the file, and on the number of different users. Since each of these factors may differ from file to file, design of a privacy system must take into account the relevant parameters. For example, a continuous key for encoding an entire file may be impractical, as entry at intermediate points would be impossible. If a complex privacy transformation is desired additional parallel hardware may be required, since direct implementation by programming may unreasonably increase the processing time. In order to provide the necessary control and integrity of the transformation system, and to meet the processing time requirements, a simple, securely housed processor similar to a common input-output control unit might be used to implement the entire file control and privacy system.

The processor

The processor and its associated random-access storage units contain the basic monitor program, system programs for various purposes, and programs and data of currently serviced users. The role of the monitor is to provide the main line of defense against infiltration attempts through the software system by maintaining absolute control over all basic system programs for input-output, file access, user scheduling, privacy protection, etc. It should also be able to do this under various contingencies such as system failures and recovery periods, debugging of system programs, during start-up or shut-down of parts of the system, etc. Clearly, the design of such a fail-safe monitor is a difficult problem. Peters[9] describes a number of principles for obtaining security through software.

Penetration attempts against the monitor or other system programs attempt to weaken its control, bypass application of various countermeasures, or deteriorate its fail-safe properties. Since it is unlikely that such penetrations could be successfully attempted from outside the processor facility, protection relies mainly on adequate physical and personnel integrity. A first step is to keep the monitor in a read-only store, which can be altered only physically, housed under lock and key. In fact, it would be desirable to embed the monitor into the basic hardware logic of the processor, such that the processor can operate only under monitor control.

Software integrity could be maintained by frequent comparisons of the current systems programs with carefully checked masters, both periodically and after each modification. Personnel integrity must, of course, be maintained at a very high level, and could be buttressed by team operation (forming a conspiracy involving several persons should be harder than undermining the loyalty of a single operator or programmer).

Integrity management procedures must be augmented with measures for controlling the necessary accesses to the privacy-protection programs; or devices for insertion of passwords, keys, and authorization lists; or for maintenance. These may require the simultaneous identification and authentication of several of the information-center personnel (e.g., a system programmer and the center "privacy manager"), or the use of several combination locks for hardware-implemented privacy-protection devices.

Hierarchy of privacy protection: Privacy protection requires a hierarchy of system-operating and privacy-protection programs, with the primary system supervisor at the top. Under this structure, or embedded in it, may exist a number of similar but independent hierarchies of individual users' privacy-protection programs. It is neither necessary nor desirable to permit someone who may be authorized to enter this hierarchy at a particular level to automatically enter any lower level. *Access should be permitted only on the basis of an authenticated "need to know."* For example, if privacy transformations are employed by a particular user, his privacy programs should be protected against access by any of the system management personnel.

Time-sharing: Various modes of implementing time-sharing in the information system may affect the privacy of information in the processor. In particular, copying of residual information in the dynamic portions of the storage hierarchy during the following time-slice seems likely. Since erasing all affected storage areas after each time-slice could be excessively time consuming, a reasonable solution may be to "tag" pages of information and programs as "private," or to set aside certain areas of the core for private information and erase only those areas or pages after each time-slice. Only the private areas or pages would need to be erased as part of the swapping operation.[14]

Also important is the effect of privacy transformations on processing time. Sophisticated privacy transformations, for example, if applied by programmed algorithms, may require a significant fraction of each time-slice. It may be necessary, therefore, to use hardware implementation of privacy transformations by including these in the hardware versions of the monitor or through the use of a separate parallel processor for all access-control and privacy-transformation operations.

Hardware failures: With respect to integrity management, there is one aspect of hardware integrity which is the responsibility of the original equipment manufacturer, viz., a hardware failure should not be catastrophic in the sense that it would permit uncontrolled or even limited access to any part of the system normally protected. Whether this entails making the critical parts of the hardware super-reliable and infallible, or whether the system can be designed for a fail-safe form of graceful degradation is an open question. It is important to assure the user that the hardware has this basic characteristic.

CONCLUDING REMARKS

It should be emphasized again that the threat to information privacy in time-shared systems is credible, and that only modest resources suffice to launch a low-level infiltration effort. It appears possible, however, to counter any threat without unreasonable expenditures, provided that the integrity and competence of key personnel of the information center is not compromised. *Trustworthy and competent operating personnel will establish and maintain the integrity of the system hardware and software which, in turn, permits use of their protective techniques.* A concise assessment of the effectiveness of these techniques in countering a variety of threats is presented in Table II.

Privacy can be implemented in a number of ways ranging from system enforced "mandatory privacy," where available privacy techniques are automatically applied to all users with associated higher charges for processing time, to "optional privacy" where a user can specify what level of privacy he desires, when it should be applied, and perhaps implement it himself. The latter privacy regime appears more desirable, since the cost of privacy could be charged essentially to those users desiring protection. For

example, if a user feels that simple passwords provide adequate protection, his privacy system can be implemented in that way, with probable savings in processing time and memory space.

Implementation of "optional" privacy could be based on a set of "privacy instructions" provided by the programming and query languages of the system. Their proper execution (and safeguarding of the associated password lists and keys) would be guaranteed by the system. The cost of their use would reflect itself in higher rates for computing time. For example, the AUTHENTICATE, K,M instruction requests the next password from list K which has been previously allocated to this user from the system's pool of password lists (or has been supplied by the user himself). The operating program now sends the corresponding message to the user who takes the correct password from his copy of the list K and sends it to the processor. If the comparison of the received password with that in the processor fails, the program transfers to location M where "WRITE LOG, A" instruction may be used to make a record of the failure to authenticate in audit log A. Further instructions could then be used to generate a real-time alarm at the processor site (or even at the site of the user's terminal), terminate the program, etc. Other privacy instructions would permit application of privacy transformations, processing restrictions, etc.

The privacy protection provided by any system could, of course, be augmented by an individual user designing and implementing privacy protection schemes within his programs. For example, he may program his own authentication requests, augmenting the system-provided instructions for this purpose; and use his own schemes for applying privacy transformations to data in files or in the communication network. Since these additional protective schemes may be software implemented they would require considerable processing time, but would provide the desired extra level of security.

Further work: This paper has explored a range of threats against information privacy and pointed out some of the systems implications of a set of feasible countermeasures. Considerable work is still needed to move from feasibility to practice, although several systems have already made concrete advances. Special attention must be devoted to establishing the economic and operational practicality of privacy transformations: determining applicable classes of transformations and establishing their work factors; designing economical devices for encoding and decoding; considering the effects of query language structure on work factors of privacy transformation; and determining their effects on processing time and storage requirements.

Large information systems with files of sensitive information are already emerging. The computer community has a responsibility to their users to insure that systems not be designed without considering the possible threats to privacy and providing for adequate countermeasures. To insure a proper economic balance between possible solutions and requirements, users must become aware of these considerations and be able to assign values to information entrusted to a system. This has been done in the past (e.g., industrial plant guards, "company confidential" documents, etc.), but technology has subtly changed accessibility. The same technology can provide protection, but we must know what level of protection is required by the user.

REFERENCES

1 *The computer and invasion of privacy*
Hearings Before a Subcommittee on Government Operations House of Representatives July 26-28 1966 U S Government Printing Office Washington DC 1966

2 P BARAN
Communications computers and people
AFIPS Conference Proceedings Fall Joint Computer Conference Part 2 Vol 27 1965

3 S ROTHMAN
Centralized government information systems and privacy
Report for the President's Crime Commission September 22 1966

4 E E DAVID R M FANO
Some thoughts about the social implications of accessible computing
AFIPS Conference Proceedings Fall Joint Computer Conference Part 1 Vol 27 pp 243-251 1965

5 S D PURSGLOVE
The eavesdroppers: fallout from R&D
Electronic Design Vol 14 No 15 pp 35-43 June 21 1966

6 S DASH R F SCHWARTZ R E KNOWLTON
The eavesdroppers
Rutgers University Press New Brunswick New Jersey 1959

7 W H WARE
Security and privacy: similarities and differences
presented at SJCC 67 Atlantic City New Jersey

8 R C DALEY P G NEUMANN
A general-purpose file system for secondary storage
AFIPS Conference Proceedings Fall Joint Computer Conference Part 1 Vol 27 p 213 1965

9 B PETERS
Security considerations in a multi-programmed computer system
presented at SJCC 67 Atlantic City New Jersey

10 P BARAN
On distributed communications: IX. security, secrecy and tamper-free considerations
RM-3765-PR The RAND Corporation Santa Monica California August 1964

11 C E SHANNON
Communications theory of secrecy systems
Bell System Technical Journal Vol 28 No 4 pp 656 715 October 1949

12 R L DENNIS
Security in computer environment

SP 2440/000/01 System Development Corporation August 18 1966

13 D J DANTINE
Communications needs of the user for management information systems
AFIPS Conference Proceedings Fall Joint Computer Conference Vol 29 pp 403-411 1966

14 R L PATRICK Private communication

Table II.

SUMMARY OF COUNTERMEASURES TO THREAT TO INFORMATION PRIVACY

Threat \ Countermeasure	Access Control (passwords, authentication, authorization)	Processing Restrictions (storage, protect, privileged operations)	Privacy Transformations	Threat Monitoring (audits, logs)	Integrity Management (hardware, software, personnel)
Accidental: User error	Good protection, unless the error produces correct password	Reduce susceptibility	No protection if depend on password; otherwise, good protection	Identifies the "accident prone"; provides post facto knowledge of possible loss	Not applicable
System error	Good protection, unless bypassed due to error	Reduce susceptibility	Good protection in case of communication system switching errors	May help in diagnosis or provide post facto knowledge	Minimizes possibilities for accidents
Deliberate, passive: Electromagnetic pick-up	No protection	No protection	Reduces susceptibility; work factor determines the amount of protection	No protection	Reduces susceptibility
Wiretapping	No protection	No protection	Reduces susceptibility; work factor determines the amount of protection	No protection	If applied to communication circuits may reduce susceptibility
Waste Basket	Not applicable	Not applicable	Not applicable	Not applicable	Proper disposal procedures
Deliberate, active: "Browsing"	Good protection (may make masquerading necessary)	Reduces ease to obtain desired information	Good protection	Identifies unsuccessful attempts; may provide post facto knowledge or operate real-time alarms	Aides other countermeasures
"Masquerading"	Must know authenticating passwords (work factor to obtain these)	Reduces ease to obtain desired information	No protection if depends on password; otherwise, sufficient	Identifies unsuccessful attempts; may provide post facto knowledge or operate real-time alarms	Makes harder to obtain information for masquerading; since masquerading is deception, may inhibit browsers
"Between lines" entry	No protection unless used for every message	Limits the infiltrator to the same potential as the user whose line he shares	Good protection if privacy transformation changed in less time than required by work factor	Post facto analysis of activity may provide knowledge of possible loss	Communication network integrity helps
"Piggy-back" entry	No protection but reverse (processor-to-user) authentication may help	Limits the infiltrator to the same potential as the user whose line he shares	Good protection if privacy transformation changed in less time than required by work factor	Post facto analysis of activity may provide knowledge of possible loss	Communication network integrity helps
Entry by system personnel	May have to masquerade	Reduces ease of obtaining desired information	Work factor, unless depend on password and masquerading is successful	Post facto analysis of activity may provide knowledge of possible loss	Key to the entire privacy protection system
Entry via "trap doors"	No protection	Probably no protection	Work factor, unless access to keys obtained	Possible alarms, post facto analysis	Protection through initial verification and subsequent maintenance of hardware and software integrity
Core dumping to get residual information	No protection	Erase private core areas at swapping time	No protection unless encoded processing feasible	Possible alarms, post facto analysis	Not applicable
Physical acquisition of removable files	Not applicable	Not applicable	Work factor, unless access to keys obtained	Post facto knowledge form audits of personnel movements	Physical preventative measures and devices

A brief description of privacy measures in the RUSH time-sharing system

by J. D. BABCOCK
Allen-Babcock Computing, Inc.
Los Angeles, California

INTRODUCTION

Brief summary of the ABC time-sharing system

Allen-Babcock's RUSH (Remote Users of Shared Hardware) comprises some 80 modules of processors operating in a time-sharing mode on an IBM SYSTEM/360, Model 50. The RUSH Monitor-Executive controls 60 simultaneous terminal users whose programs reside in a large bulk store (2,097,152 bytes-directly addressable). RUSH co-exists with Operating System/360, option 2 (multi-programming, fixed tasks) and uses the file management schemes of this operating system.

Each RUSH user converses with an IBM 2741 Selectric terminal or a Teletype Model 28, 32, 33, 35 or 37. RUSH supports both scientific and commercial applications, the latter characterized by problems needing general character manipulation and file management at the higher language level. The only conversational language offered by RUSH is a problem-oriented version of PL/I; and users range from the casual to the professional programmer.

File security in RUSH

An early design decision was to build "protection" software for the RUSH monitor, using the basic facilities of Data Management processors in OS/360, since IBM was not planning to implement their security options in early distributions.

First, unauthorized access to RUSH during user Sign-on had to be blocked. The LOGIN statement contains three levels of protection:

(1) Master Account Identification;
(2) Sub-Account Identification;
(3) Protection Key on (2).

The second try at an invalid Master Account causes RUSH to disconnect the telephone line. Also, if both the Master Account and the Sub-Account pass, the protect key—if improperly returned—causes a disconnect.

The second area requiring protection is user program and data files. The object of a %LOAD/SAVE statement is 1 or 2 parameters:

(NAME, KEY).

The user supplies the name and can optionally add the KEY. The first four characters of the KEY are for 'read only' requests, and the full six characters must be supplied in order to write over the copy of NAME on the disk. All of the LOAD/SAVE area is under protection of the master account name. Since the LOAD/SAVE command cannot be executed in a computer loop, it is extremely difficult to ascertain the full KEY by trial-and-error.

The third protection mechanism is embodied in the RJE (Remote Job Entry) mode. RJE allows the user to build a job stream for eventual compilation and execution in a separate partition of memory. It scans all control cards and limits names of files to a very narrow range, i.e., the customer and his file names must agree in part with his master account number in the LOGIN statement. RJE allows execution of ABC approved catalogue procedures *only*. An RJE user cannot read or write files other than his own.

Finally, RUSH uses the full System/360 memory-protection feature. Memory can be protected in blocks and only legal blocks for a particular active user may be processed by the Executive. Further, the Executive is operated under a key that disallows all "store" commands within itself; thus it cannot destroy itself. RUSH is fully interpretative, affording further protection for its users.

Possible threats and what can be done

RUSH has no defense against hardware penetrations such as wiretaps. However, attempted penetrations are logged after forcing a line disconnect. In the every near future users will have a "call-back" system for very sensitive files. When a file is opened by the program, OS/360 will check the header label for

authorized access; if needed, the operator is informed and the user receives a message to telephone the password to the operator on a separate number. The operator then instructs the system to perform the open rator then instructs the system to perform the open or refuse. The legal user can alter the password hourly, daily, weekly, etc. by informing the Data-Center.

An advantage of remote systems against penetration attempts is that the data-center room is sealed from on-site penetration, a feature not found in traditional batch shops. Remote systems do not allow the opportunity for unauthorized access to the central computer facility. Another advantage is the complete "READ" protection offered by System/360. The software for this feature is not difficult, but must be supported by each installation to its particular requirements. Thus, memory partitions can not be penetrated.

A final means of protection is to use a system which is *completely* interpretative, such that any penetration can be detected; user-user, as well as user-monitor-user. This protection is now available in RUSH, and guarantees full security to time-sharing users, program, data, and program/data files. Further, RJE users are limited to non-machine language processors available in OS/360.

SUMMARY

Teletypewriters offer non-printable characters which can be used as passwords that are never printed. However, the deliberate leakage of one user's Master Account Identification to another is difficult to control. Protection for RUSH customers is achieved by changing this ID frequently. Graphic devices probably offer no more problems than do current typewriters. Passwords could be transmitted with no visual answer-back.

RUSH has never "lost" any files inadvertently, attesting to its overall design and its local KEY system. Also, no deliberate penetrations have so far been detected.

A brief description of privacy measures in the multics operating system*

by EDWARD L. GLASER
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

The problem of maintaining information privacy in a multi-user, remote-access system is quite complex. Hopefully, without going into detail, some idea can be given of the mechanisms that have been used in the Multics operating system at MIT.

The heart and soul of any on-line system is the file subsystem, which is charged with the maintenance of all on-line data bases and with their safekeeping from accidental or malicious damage. The salient features of the Multics file system* are: (1) all references to data are by symbolic name and never by physical address; (2) associated with each file or substructure within the system is an access-control list that defines each authorized user and how he may gain access to the substructure or file. (User class identifiers may be employed as well as individual user names.) Because this file-system mechanism also safeguards much of the operating system itself, substructures of directories and files are used to store the system and the state of individual user programs during execution. There is no other special "swapping" on-line file structure.

In order to make the file system effective, each user must carry an identification. This identification is made when he logs into the system, and is held as part of the user data base during the logged time period. Determining that a user is who he says, is accomplished by means of a log-in routine which may include passwords, special log-in algorithms, etc.

Even with the protection supplied by the file system, a central portion of the supervisor must be protected against accidental or overt tampering. A combination of hardware and software means to prevent gaining an unusual privilege is employed. In part, these safeguards include hardware locks that prevent execution, reading, or modification of certain key portions of the supervisor except when responding to a generated interrupt. (Locks that are more complex than can be explained in this short account are also employed.) Two basic principles are applied within this part of the supervisor:

(1) *Compartmentalization*—functionally separating the various activities of the supervisor into program modules. Since it is assumed that there will be accidents, or that individuals may on occasion be able to thwart the supervisor, compartmentalization insures a minimum amount of damage or a minimum loss of privileged information.
(2) *Auditability*—By properly identifying each module of the operating system, so that it is possible at any time to determine that the system currently running is the intended one.

It is also possible to include the ability to determine whether a user has violated the system, and if he has, to observe his malpractice by means of a special high-privilege system function. The latter mechanism includes an additional set of safeguards that provide an audit trail indicating the observer, who was observed, and the date of observation. This auditing information is recorded

*Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advance Research Projects Agency, Department of Defense under Office of Naval Research Contract Number Nonr-4102(01).

*The Multics file system is described by R. C. Daley and P. G. Neumann in "A General-Purpose File System for Secondary Storage," presented at the 1965 Fall Joint Computer Conference (AFIPS Conf. Proc., Vol. 27, Part 1).

in such a way as to make it extremely difficult for one or two individuals to destroy it without being observed.

The maintenance of privacy and informational integrity within systems that are still not fully comprehended is a large and intricate problem. The present discussion suggests only some key considerations.

The influence of machine design on numerical algorithms*

by W. J. CODY
Argonne National Laboratory
Argonne, Illinois

INTRODUCTION

A great deal has been said and written recently about the poor arithmetic characteristics of computer designs,[1,2,3,4,5] largely because anomalies long present in the floating point arithmetic of binary computers have suddenly been magnified in importance by the appearance of base 16 arithmetic. This paper is not intended as an attempt to influence present or future machine design—we will leave that project to those who think they *can* influence such designs. Rather, it is written in the spirit of H. R. J. Grosch when he advised programmers "to go back to work; quit trying to remodel the hardware ... accept reality. ... "[6]

It is our purpose to examine the "reality" of some of the most common design features of recent computers and how they affect the implementation of numerical algorithms. To do this we will consider the floating point arithmetic units on two commercial computers, the Control Data 3600 and the IBM System/360, used by the Applied Mathematics Division at Argonne National Laboratory. These two machines are conventional in design, yet offer great variety and contrast in those arithmetic design characteristics we wish to discuss.

Both machines have built-in single and double precision floating point. The CDC machine, however, operates with exponent base 2 while the IBM machine uses exponent base 16. Of course, both machines treat the mantissa separately from the exponent in floating point operations, but the 3600 rounds the mantissa prior to a renormalization shift while the 360 truncates prior to renormalization in double precision and after renormalization in single precision. Renormalization shifts in the 3600 are in multiples of one bit while those in the 360 are in multiples of four bits. Thus, after multiplication of two normalized numbers the 3600 has at most a one bit shift for renormalization while the 360 shifts either zero or four bits.

There are other aspects of these machines that are important in detailed numerical work but we will concern ourselves only with the availability of double precision, the effects of rounding (including number truncation) and the renormalization shift.

Effect of renormalization

One of the first things one must realize in detailed floating point numerical computation is that computers deal with a finite discrete subset of the real number system using arithmetic that does not always obey some of the usual rules; e.g., the associative and distributive laws may not hold. Most of the trouble occurs in the process of truncation or rounding of results coupled with possible renormalization shifts.

As an indication of what can happen, let us show that three of the four possible modes of floating point arithmetic on the computers under discussion do not allow a multiplicative identity in the mathematical sense of the term. For illustrative purposes, assume the arithmetic is single precision as incorporated in the CDC 3600, but assume only four bits in the mantissa of a number. Let x = 1.125 and consider the product 1.0 × x. Then (the mantissas below will be expressed as binary fractions)

$$x = 2^1 \times .1001$$
$$\underline{\times\ 1.0 = 2^1 \times .1000}$$
$$2^2 \times .0100\ 1$$

The result mantissa is now rounded to give

$$2^2 \times .0101$$

and then renormalized by a left shift of one unit with an appropriate adjustment of the exponent to give

$$2^1 \times .1010 = 1.250.$$

*Work performed under the auspices of the U. S. Atomic Energy Commission.

Thus this commonly used rounding scheme when employed prior to the renormalization shift precludes the existence of a multiplicative identity.

If the rounding process were replaced by simple truncation, but still employed prior to renormalization (as happens on the CDC machine if rounding is suppressed and on the IBM machine in double precision), the above result would be

$$2^1 \times .1000 = 1.000.$$

Again unity is not a true multiplicative identity. Only the IBM single precision mode with its four guard bits and truncation after renormalization guarantees that

$$1.0 \times x = x$$

for all x.

The important point here is not the lack of a multiplicative identity in the mathematical sense, but that predictable arithmetic errors have occurred. A little thought will show that the same erroneous results can occur any time one of the factors is an exact power of 2. However, in the CDC machine the error is always limited to the last bit while it will involve the last four bits when it occurs on the system/360. The replacement of the expression $2.0 \times x$, for instance, by $x + x$ saves on the average about ½ bit per operation on the CDC machine, but saves almost 2 bits per operation on the IBM machine in double precision. Thus the discrepancy present in most binary machines is magnified by the use of base 16 arithmetic.

The use of four guard bits in the IBM single precision mode effectively corrects the problem of renormalization in multiplication. Note, however, that single and double precision arithmetic no longer obey the same rules, hence there may be Fortran programs, for example, in which conversion from one precision to another will alter the arithmetic behavior.

Consider next the use of Newton-Raphson iteration for the square root. Let A be the square root of x, let A_0 be an initial estimate of A and use the usual iteration scheme

$$A_n = \tfrac{1}{2}(A_{n-1} + x/A_{n-1}) \qquad n = 1,2,\ldots$$

to obtain successively better estimates. This is known to be a convergent scheme with $\{A_n\}$ approaching A monotonically from above for $n > O$. If the iteration is continued until two successive estimates are equal, no further improvement is theoretically possible and we have the best estimate of A. But let us follow the iteration in detail with System/360-type arithmetic (we will use only 12 bits in the mantissa) assuming we are near convergence and A_{n-1} has a mantissa greater than ½. Say

$$\begin{aligned} A_{n-1} &= 16^m \times .1010\ 1011\ 0000 \\ +\ x/A_{n-1} &= 16^m \times .1010\ 1010\ 1010 \\ &\overline{\qquad\qquad 16^m \times 1.0101\ 0101\ 1010.} \end{aligned}$$

The mantissa is now shifted right four places to give

$$16^{m+1} \times .0001\ 0101\ 0101\ 1010$$

and the last four bits are truncated. (There go the single precision guard bits!) Now multiplication by ½ gives

$$\begin{aligned} & 16^{m+1} \times .0001\ 0101\ 0101 \\ \times\ 1/2 = & 16^0 \quad \times .1000\ 0000\ 0000 \\ & \overline{\qquad\qquad 16^{m+1} \times .0000\ 1010\ 1010\ 1000} \end{aligned}$$

which becomes

$$16^m \times .1010\ 0101\ 1000$$

upon renormalization if single precision is used, and

$$16^m \times .1010\ 1010\ 0000$$

if double precision is in use. Note that the last three bits are of necessity zero in single precision in spite of the existence of the guard gits. Not only are more accurate estimates of A possible, but it may be that $A > A_n$ with this method, contrary to the theory.

Rewriting the iteration in the form

$$A_n = A_{n-1} + \tfrac{1}{2}(x/A_{n-1} - A_{n-1})$$

for at least the final iteration clears up the problem.

The original iteration scheme works quite well on the CDC machine, since the normalization shift after the addition involves only one bit and the multiplication by ½ is accomplished with no error by an alteration of the exponent in the results.

Truncation vs Rounding

As we have already seen, the four guard bits in the System/360 single precision operations offer protection in the case of multiplication. They offer no protection, however, against the build-up of truncation errors, even in short computations. Although this phenomenon is well known the following example of its effect may be of some interest to those who have never before witnessed it.

Consider the computation of

$$y = \int_0^1 x/3\ dx$$

by means of trapezoidal quadrature using a mesh size $h = 1/n$, $n = 2,4,8,\ldots,256$. That is, approximate y by the computation

$$h \sum_{j=0}^{n}{}'' (j \times h)/3$$

where the double prime indicates only half of the first and last terms are to be considered. It will become obvious that one would never carry out this particular computation in the manner we will describe, nor for the large values of n we will use. The justifications for this example are that it is not far from being a practical problem (one need only change the integrand) and it is possible to almost completely isolate the effect of truncation.

We will actually construct the (j×h)/3 for the various values of j and sum them. For the choices of h made here the quantities j × h can always be expressed exactly as machine numbers, eliminating one possible source of error. Theoretically trapezoidal quadrature should give an exact answer for a linear integrand, eliminating a second source of error. Finally, if the integrand is computed in double precision as needed and then converted to single precision, the only remaining sources of error are possible inaccuracies in conversion to single precision and the summation process itself. Table I lists the results of the computation carried out with Fortran codes in single precision arithmetic, with the exception noted above, on both machines. The results are listed in octal and hexadecimal format to avoid the error contamination involved in conversion to decimal form. Note that the results on the CDC machine never stray too far from the correct results even when n becomes fairly large, whereas the System/360 results become progressively worse as n increases. Repeating the experiment in complete double precision, nullifying the possible effect of computation of the integrand in a higher precision, made no essential change in the low order bit patterns.

TABLE I

Trapezoidal Quadrature Applied to $\int_0^1 x/3\ dx$ in Single Precision Floating Point on CDC 3600 and IBM System/360

No. of Points	*CDC 3600*	*IBM System/360*
2	1775525252525254	402AAAAA
4	1775525252525254	402AAAAA
8	1775525252525252	402AAAA8
16	1775525252525254	402AAAA8
32	1775525252525252	402AAAA7
64	1775525252525254	402AAAA6
128	1775525252525252	402AAA9A
256	1775525252525252	402AAA87
Correct Value	1775525252525253	402AAAAA

For this particular example the IBM arithmetic shows up quite badly. In view of the short word length (24 bit mantissa) in single precision it is particularly important to use an algorithm that will minimize the accumulation of truncation error. Examples can be concocted which show that rounding errors can also build up quite badly, but rounding is a much less biased operation than truncation and is therefore not as likely to cause serious deterioration of results. It is also the author's feeling that effective remedial action on the part of the programmer, e.g., the rearrangement of computations so as to minimize error build-up, is easier when the machine arithmetic uses rounding than it is when the arithmetic uses truncation.

Double precision

One feature of recent computers has simplified rather than complicated the work of a numerical analyst—the inclusion of inexpensive (timewise) double precision arithmetic. The advantage of double precision accumulation of inner products in certain matrix problems is well known, as is the use of double precision accumulation of the corrections to the dependent variable in certain methods for the solution of differential equations.[7] But there is one rather important usage of double precision which has been neglected in the literature.

Let us consider the design of a subroutine for the exponential function

$$y = e^x.$$

We recognize at the start that an argument fed to this routine will probably be in error because it is a machine number of finite length. Let Δy be the absolute error in y, and

$$\delta y = \Delta y / y$$

be the relative error. For the exponential function,

$$\delta y = x \delta x$$

and a rounding error of ½ unit in the least significant bit of x for $x \approx 80$ can be expected to lead to an error of 40 units in the least significant bit of y. Many exponential routines yield errors of this order of magnitude even when the value of x is exactly 80, i.e., even when $\delta x = 0$.

The inaccuracy lies in the argument reduction scheme usually used. We will assume we are using the CDC machine with its base 2 arithmetic in what follows, although this assumption is not crucial. A typical exponential routine then proceeds somewhat as follows. Let

$$n = [x/\ell n\ 2 + \tfrac{1}{2}]$$

where [a] denotes the "integer part of a," and let

$$g = x - n \cdot \ell n\, 2.$$

Then

$$e^x = 2^n \cdot e^g,$$

reducing the problem of computing any exponential to that of computing the exponential of a reduced argument g where $|g| \leq \ell n\, 2/2$. This last can generally be done quite accurately provided g is accurate. The greatest loss in accuracy occurs in the subtraction involving the two nearly equal quantities x and $n \cdot \ell n\, 2$. When these quantities agree for the first m significant bits, g may be in error in the last m significant bits leading to a large value of δg independently of whether δx is zero or not.

It is precisely here that double precision operations can save the day. Assume $\delta x = 0$, hence x can be converted to a double precision number with no introduced error. Since $\ell n\, 2$ is a constant known to any precision desired, the produce $n \cdot \ell n\, 2$ can be found in double precision and the value of g determined quite accurately at a total cost of one double precision multiplication and one double precision subtraction. The cost of converting the single precision variables x and n to double precision and of converting g back again is generally quite small when programming is done in assembly language. This last conversion could even be simple truncation of the mantissa. Since $|g| < .7$, an error of 1 unit in the last bit of g will propagate at most as .7 unit in the last bit of e^g.

The payoff for this technique can be quite large. Table II compares the results of computations of e^x for x = 80(1)100 with routines differing only in the argument reduction scheme. In a systems library the accuracy obtainable in this manner is of utmost importance. Not only do users regard systems routines as "black boxes" which return correct results, but other systems routines do as well. It is not at all uncommon for an exponential routine to be called by the system routines for complex trigonometric functions as well as by the power routine, i.e., the routine to evaluate the Fortran expression x**y. The basic CDC 3600 library developed at Argonne National Laboratory uses this argument reduction technique in almost all of the real and complex trigonometric routines as well as the power routine. Computations with these routines rival in accuracy computations made by converting the independent variable to double precision and using the appropriate double precision routine. It is generally even more accurate to evaluate x**I by converting I to floating point and using the floating point power routine than it is to use the integer power routine directly.

Single precision floating point operations that produce double length results, as in the CDC machine, can also be quite useful. Consider the computation of

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

Table II

Error in units of least significant bit for computation of e^x on a CDC 3600 in single precision floating point. An Error of -20 indicates the computed result was 20_{10} units low

x	*Error using standard argument reduction*	*Error using modified argument reduction*
80	48	0
81	54	0
82	6	−1
83	−16	0
84	−28	0
85	76	1
86	35	0
87	27	−1
88	7	−1
89	−16	−1
90	−47	0
91	57	0
92	54	1
93	19	0
94	3	1
95	−13	0
96	85	−1
97	87	0
98	39	−1
99	27	0
100	0	0

for large values of x by means of the asymptotic expansion

$$\operatorname{erfc}(x) \approx \frac{1}{\sqrt{\pi}} \frac{e^{-x^2}}{x} \left\{ 1 + \sum_{m=1}^{\infty} \frac{(2m-1)!!}{(2x^2)^m} \right\}$$

where $(2m-1)!! = \dfrac{(2m)!}{2^m (m)!} \begin{cases} 1 \cdot 3 \cdot 5 \cdots (2m-1) & \text{for } m > 0, \\ 1 & \text{for } m = 0. \end{cases}$

For x large enough the main source of error lies in the evaluation of e^{-x^2}. Even if x is considered to be exact, i.e., $\delta x = 0$, the product x^2 used as an argument for the single precision exponential routine has an error in the least significant bit due to rounding. Double

precision argument reduction within the exponential routine cannot save the computation, but an analogous reduction before entry to the exponential routine can. The quantity x^2 can be formed accurately to double precision by either converting x to double precision and squaring, or by squaring x in single precision while suppressing the rounding and renormalization of the double length product. Computation of the reduced argument g can then proceed as outlined above, and it is this reduced argument which is transmitted to the exponential routine.

SUMMARY

Through the use of some elementary examples we have shown that the design of the arithmetic unit of a computer can influence the design of accurate numerical subroutines. The author firmly believes that the design of an algorithm does not end with its presentation in an algebraic language, but that it must change in detail as it is implemented on various machines.

REFERENCES

1 ROBERT T GREGORY
On the design of the arithmetic unit of a fixed word length computer from the standpoint of computational accuracy
IEEE Transactions on Electronic Computers EC 15 p 255 1966

2 LEONARD J HARDING JR
Idiosyncracies of system 360 floating point
presented at SHARE XXVII Toronto Canada 1966

3 R W HAMMING W KAHAN H KUKI C L LAWSON
Panel discussion on machine implementation of numerical algorithms
Presented at SICNUM meeting Los Angeles California 1966

4 R MONTEROSSO
Differenze nella rappresentazione e nella arithmetica floating tra le serie IBM 700 7000 E la serie 360 errori di arrotondamento nelle operazioni elementari floating point dei calcolatori della serie IBM 360
Report Eur 2777 i Comunita Europea Del'Energia Atomica Ispra Italy April 1966

5 JOHN H WELSCH
Proposed floating point feature for IBM system 360
presented at SHARE XXVII Toronto Canada 1966

6 H R J GROSCH
Programmers: The industry's Cosa Nostra
Datamation 12 p 202 1966

7 RICHARD KING
Reduction of Roundoff Error in Gill Runge Kutta Integration Argonne National Laboratory Applied Mathematics Division Technical Memorandum No 22 Argonne Illinois June 1961

Base conversion mappings*

by DAVID W. MATULA
Washington University
St. Louis, Missouri

INTRODUCTION AND SUMMARY

Most computers utilize either binary, octal or hexadecimal representation for internal storage of numbers, whereas decimal format is desired for most input, output and for the representation of constants in a user language such as FORTRAN or ALGOL. The conversion of a normalized number in one base to a normalized number in another base is beset with difficulties because of the incompatibility of exact representations of a number when only a limited number of digits are available in each base.

Specifying conversion either by truncation (sometimes called chopping) or by rounding will define a function mapping the numbers representable with a fixed number of digits in one base into the numbers representable with some fixed number of digits in another base. Practical questions which immediately arise are:

(1) Is the conversion mapping one to one, i.e., are distinct numbers always converted into distinct numbers?

(2) Is the conversion mapping onto, i.e., is it possible to achieve every distinct number representable in the new base by the conversion of some number representable in the old base?

Excluding the case where both bases are powers of some other base (e.g., octal and hexadecimal), we show that a conversion mapping, whether by truncation or rounding, can never be both one-to-one and onto. Specifically, our major result, the Base Conversion Theorem, says: Given a conversion mapping of the space of normalized n digit numbers to the base β into the space of normalized m digit numbers to the base ν, where $\beta^i \neq \nu^j$ for any non-zero integers i, j, the conversion mapping via truncation or rounding

(1) is one-to-one if and only if

$$\nu^{m-1} \geq \beta^n - 1$$

(2) is onto if and only if

$$\beta^{n-1} \geq \nu^m - 1$$

The necessary condition for the existence of a one-to-one conversion intuitively appears overly severe, as the condition $\nu^m \geq \beta^n$ is sufficient to assure a one-to-one conversion of all integers from 1 to β^n. However, there are essential difficulties of numerical representation to different bases *not* related to the nature of the conversion method which indeed make this more restrictive bound necessary. From a practical viewpoint these difficulties will often occur for numbers of a reasonable order of magnitude, i.e., well within the limitations on exponent size required for the normalized number stored in a computer word. For example, conversion of normalized four digit decimal numbers into normalized four digit hexadecimal numbers is not one-to-one, and to see this consider the number 65,536 (i.e. 64K, which is certainly not considered too large by most computer users), and note that

$$.1000_{16} \times 16^5 = 65{,}536 < .6554 \times 10^5 <$$
$$.6555 \times 10^5 < 65{,}552 = .1001_{16} \times 16^5.$$

Thus the two distinct four digit decimal numbers $.6554 \times 10^5$ and $.6555 \times 10^5$ will be converted by trucation into the same four digit hexadecimal number $.1000_{16} \times 16^5$. The necessity of having five hexadecimal digits to distinguish $.6554 \times 10^5$ from $.6555 \times 10^5$ is surprising when it is observed that the four digit decimal integers 1000 through 4095 may all be converted exactly utilizing just three hexadecimal digits.

The conditions of the Base Conversion Theorem assert the impossibility in general of a decimal to binary or binary to decimal mapping being both one-to-one and onto. Thus from a practical viewpoint, some compromise must be made when determining the number of significant digits allowable in the normalized decimal constants of a user language given a fixed size binary (or octal or hexadecimal) word available in some computer hardware. This problem is discussed in the final section.

*This research was performed under USPHS Research Contract No. ES-00139-01.

Significant spaces and normalized numbers

The notion of normalized number relates to how a number may be represented. Care must be taken in forming a precise definition of a space of normalized numbers so that a number is not confused with the representation of the number; i.e., in defining the space of three digit normalized decimal numbers, we do not wish to include ".135" and at the same time exclude ".1350," as they indeed represent the same real number. Hence we wish to think of the "space of n digit normalized numbers to the base β" not as a set of representations, but as a *set of real numbers* which may be given a specified form of representation.

Definition: For the integers $\beta \geq 2$, called the *base*, and $n \geq 1$, called the *significance*, let S^n be the following set of real numbers

$$S_\beta^n = \left\{ x \;\middle|\; |x| = \sum_{i=1}^{n} \alpha_i \beta^{j-i},\ j,\ \alpha_i \text{ integers},\ 0 \leq \alpha_i \leq \beta-1 \text{ for } 1 \leq i \leq n \right\}$$

What we have called the significant space, S_β^n, may be interpreted as the space of normalized β -ary numbers of n significant β -ary digits along with the number zero, since the elements of S_β^n other than zero are exactly those numbers which may be given a normalized n β -ary digit representation. We have tacitly assumed no bound on the exponent but the possible effects of bounding the exponent size are considered in the final section.

Using the natural ordering of the real numbers every number in S_β^n other than zero will have both a next smallest and next largest neighbor in S_β^n. These numbers play an important part in the discussion of conversion, so that we introduce the following notation.

Definition: For $x \in S_\beta^n$, $x \neq 0$, x' is the *successor* of x in S_β^n if and only if

$$x' = \min \{y \mid y > x,\ y \in S_\beta^n\}$$

To avoid ambiguity we occasionally use $[x]'$ for x'.

Examples:

In S^4_{10} we have

$$5123' = 5124$$

$$\left[-.18 \times 10^{15}\right]' = -.1799 \times 10^{15}$$

$$9.999' = 10$$

$$70700' = 70710$$

Corollary 1: In S_β^n for any integer i

$$\left[\beta^i (1-\beta^{-n})\right]' = \beta^i$$

$$\left[\beta^i\right]' = \beta^i (1+\beta^{1-n})$$

Corollary 2: For $x \in S_\beta^n$ and any integer i

$$\beta^i \left[x\right]' = \left[\beta^i x\right]'$$

The relative difference between a number in S_β^n and its successor is a measure of the accuracy with which arbitrarily chosen real numbers may be approximated in S_β^n. It is desirable to have this relative difference depend on the magnitude of the numbers involved and not on their sign of course they must both have the same sign).

Definition: The *gap* in S_β^n at $y \in S_\beta^n$ for $y \neq 0$ is denoted by $\gamma_\beta^n(y)$ where $\gamma_\beta^n(y) = (y' - y)/y$ for $y>0$

$$\gamma^n(y) = \gamma^n(-y) \qquad \text{for } y<0$$

When there is no ambiguity $\gamma(y)$ may be used for $\gamma_\beta^n(y)$. Although the difference $y'-y$ grows monotonically with $y \in S_\beta^n$, the gap function, $\gamma_\beta^n(y)$, varies between fixed bounds for all $y \neq 0$.

Lemma:

$$\min \{\gamma_\beta^n(y) \mid y \in S_\beta^n,\ y \neq 0\} = 1/(\beta^n-1)$$

$$\max \{\gamma_\beta^n(y) \mid y \in S_\beta^n,\ y \neq 0\} = 1/\beta^{n-1}$$

Proof: For any $y \in S_\beta^n$, $y > 0$, a j can be chosen such that

$$y = \alpha_1\beta^{j-1} + \alpha_2\beta^{j-2} + \ldots + \alpha_n\beta^{j-n}$$

where $\alpha_1 \neq 0$, $0 \leq \alpha_i \leq \beta-1$ for $1 \leq i \leq n$.

Clearly $y' - y = \beta^{j-n}$, so that

$$\gamma^n{}_\beta(y) = (y' - y)/y = \beta^{j-n}/(\alpha_1\beta^{j-1} + \alpha_2\beta^{j-2} + \ldots + \alpha_n\beta^{j-n}) = 1/(\alpha_1\beta^{n-1} + \alpha_2\beta^{n-2} + \ldots + \alpha_n)$$

The minimum on the right occurs when $\alpha_i = \beta-1$ for all i, and the maximum occurs when $\alpha_1 = 1$, $\alpha_2 = \alpha_3 = \ldots \alpha_n = 0$.

Thus $1/\beta^{n-1} \leq \gamma_\beta^n(y) \leq 1/(\beta^n-1)$ for all $y \in S_\beta^n$, $y > 0$, and since $\gamma_\beta^n(-y) = \gamma_\beta^n(y)$ the bounds hold for all $y \in S_\beta^n$, $y \neq 0$.

Note that

$$\gamma_\beta^n(\beta^n-1) = 1/(\beta^n-1)$$

and

$$\gamma_\beta^n(\beta^{n-1}) = 1/\beta^{n-1}$$

Therefore both bounds are achieved and the lemma is proved.

Definition: The minimum gap $g(S_\beta^n)$ and the maximum gap $G(S_\beta^n)$ are given by

$$g(S_\beta^n) = \min \{\gamma_\beta^n(y) \mid y \in S_\beta^n, y \neq 0\}$$

$$G(S_\beta^n) = \max \{\gamma_\beta^n(y) \mid y \in S_\beta^n, y \neq 0\}$$

Thus from the Lemma

$$g(S_\beta^n) = 1/(\beta^n - 1)$$

$$G(S_\beta^n) = 1/\beta^{n-1}$$

Observe that for $y \in S_\beta^n$, $y \neq 0$, the difference $y'' - y'$ is either the same as $y' - y$ or differs by a factor of β, and that on a log-log scale all points of the graph of the gap function will lie on a saw tooth function. Figure 1 shows the saw tooth functions corresponding to $\gamma_{10}^4(y)$, $\gamma_{10}^6(y)$, $\gamma_{16}^6(y)$, and $\gamma_2^{15}(y)$. Note that there are much bigger jumps in the gap function with a larger base, so that a binary system comes closest to having uniform precision.

It is reasonable to expect that a sufficient condition for a base conversion mapping to be one-to-one is that the minimum gap in the old base be at least as large as the maximum gap in the new base. Figure 1 suggests that if this condition does not hold and if the gap function of the old base does not share some common period with the gap function of the new base, then there will exist some range of numbers where the gap function in the new base is larger than the gap function in the old base. If this region is sufficiently long, a conversion mapping of the old base into the new base could not be one-to-one over this region. To confirm these observations we begin by proving a number theoretic theorem which is of independent interest.

A number theoretic result

Theorem 1:

If $\beta, \nu > 1$ are integers such that $\beta^i \neq \nu^j$ for any positive integers i, j, then the space of rational numbers of the form β^i/ν^j, where i, j are positive integers, is dense in the positive real line. That is

(*) for any $\alpha \geq 0$, $\epsilon > 0$, there exist $i,j > 0$ such that

Proof:

$$\alpha < \beta^i/\nu^j < \alpha + \epsilon$$

It is sufficient to prove (*) for $\alpha \geq 1$, since interchanging the roles of β and ν will then verify (*) for $0 \leq \alpha \leq 1$. Furthermore it is now shown that the validity of (*) for $\alpha = 1$ implies the result for all $\alpha \geq 1$.

Given $\alpha \geq 1$, $\epsilon > 0$, let $\epsilon' = \epsilon/\alpha$ and assume $m, n > 0$ can be chosen such that

$$1 < \beta^m/\nu^n < 1 + \epsilon'$$

Then $k \geq 1$ may be determined so that

$$(\beta^m/\nu^n)^k > \alpha$$

Therefore $(\beta^m/\nu^n)^{k-1} \leq \alpha$

$$\alpha < (\beta^m/\nu^n)^k < (\beta^m/\nu^n)^{k-1}(1 + \epsilon') \leq \alpha(1 + \epsilon') = \alpha + \epsilon$$

Hence with $i = k \cdot m$, $j = k \cdot n$

$$\alpha < \beta^i/\nu^j < \alpha + \epsilon$$

Thus to prove the theorem we need only show that

(**) for any $\epsilon > 0$ there exist $i, j > 0$

such that

$$1 < \beta^i/\nu^j < 1 + \epsilon.$$

Let $\delta = \inf \left\{ \beta^i/\nu^j \;\middle|\; \beta^i \geq \nu^j, i,j>0 \right\}$

Assume $\delta > 1$. Then $k \geq 1$ may be determined to satisfy

$$\delta^k \geq \nu$$

$$\delta^{k-1} < \nu$$

Since then $\delta^k < \nu\delta$, the definition of δ assures the existence of $m, n > 0$ and $\delta' \geq \delta$ where

$$\delta' = \beta^m/\nu^n$$

Thus $(\delta')^k < \nu\,\delta$

$$\beta^{mk}/\nu^{nk+1} = (\delta')^k/\nu \geq \delta^k/\nu \geq 1$$

and

$$\beta^{mk}/\nu^{nk+1} = (\delta')^k/\nu < \delta$$

But this contradicts the definition of δ.

Therefore $\delta = 1$.

It is not possible to have $i, j > 0$ for which the limit $\delta = 1 = \beta^i/\nu^j$ is achieved, since $\beta^i \neq \nu^j$ for $i, j > 0$ by assumption in the theorem. Therefore (**) must hold, proving the theorem.

For completeness we include the following

Corollary:

If $|\beta|, |\nu| > 1$ are integers such that $|\beta|^i \neq |\nu|^j$ for

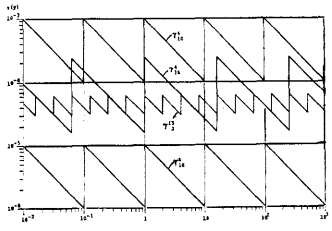


Figure 1 – Gap functions

any non-zero integers i, j, and if either β or ν or both are negative then the space of rational numbers of the form β^i/ν^j, where i, j are positive integers, is dense in the real line.

Proof:

Apply theorem 1 to $b = \beta^2$ and $v = \nu^2$ to show density in the positive real line for the rationals b^i/v^j. Then either multiplication by β (or division by ν) of the fractions b^i/v^j will show the density in the negative reals.

Note that the conditions $\beta^i \neq \nu^j$ for any non zero integers i, j, is equivalent to saying that β and ν are not both powers of some common base. This condition is clearly necessary for theorem 1 to hold as otherwise we would only be able to generate powers of this common base. The fact that this theorem can hold even when β divides ν (or when ν divides β) separates this result from a major portion of number theoretic results where a g.c.d. of unity is an essential condition.

Conversion mappings

Since the two spaces S_β^n and S_ν^m are both sets of real numbers, elements of these spaces may be compared with each other by the natural ordering on the real numbers. Utilizing the successor function we now define the conversion mappings.

Definition: The *truncation conversion map* of S_β^n into S_ν^m, $T: S_\beta^n \to S_\nu^m$, is defined as follows:

for $x \in S_\beta^n$, $y \in S_\nu^m$

$T(x) = y$ for $x > 0$, $y \leq x < y'$

$T(x) = y'$ for $x < 0$, $y < x \leq y'$

$T(0) = 0$

Definition: The *rounding conversion map* of S_β^n into S_ν^m, $R: S_\beta^n \to S_\nu^m$, is defined by:

for $x \in S_\beta^n$, $y \in S_\nu^m$

$R(x) = y'$ for $x > 0 (y + y')/2 \leq x < (y' + y'')/2$

$R(x) = y'$ for $x < 0 (y + y')/2 < x \leq (y' + y'')/2$

$R(0) = 0$

Note: The above definitions are for the usual truncation and rounding by magnitude with sign appended, so that $T(-x) = -T(x)$ and $R(-x) = -R(x)$ for all $x \in S_\beta^n$. Assuming a truncation or rounding by algebraic size (for which the above symmetry does not hold) would not change the main body of our results, but would require more tedious proofs, i.e., conversion of negative numbers would have to be treated seperately in the proofs.

With these preliminaries we are now ready to state and prove the major result.

Base Conversion Theorem

The truncation (rounding) conversion map

$$T: S_\beta^n \to S_\nu^m \quad (R: S_\beta^n \to S_\nu^m), \text{ where } \beta^i \neq \nu^j$$

for any non-zero integers i, j,

(1) is one-to-one if and only if
$$\nu^{m-1} \geq \beta^n - 1$$

(2) is onto if and only if
$$\beta^{n-1} \geq \nu^m - 1$$

Proof:

The theorem will be divided into eight parts.

(i)	$\nu^{m-1} \geq \beta^n - 1$	$\Rightarrow$ T:	$S_\beta^n \to S_\nu^m$	is one to one
(ii)	$\nu^{m-1} < \beta^n - 1$	$\Rightarrow$ T:	$S_\beta^n \to S_\nu^m$	is not one to one
(iii)	$\nu^{m-1} \geq \beta^n - 1$	$\Rightarrow$ R:	$S_\beta^n \to S_\nu^m$	is one to one
(iv)	$\nu^{m-1} < \beta^n - 1$	$\Rightarrow$ R:	$S_\beta^n \to S_\nu^m$	is not one to one
(v)	$\beta^{n-1} \geq \nu^m - 1$	$\Rightarrow$ T:	$S_\beta^n \to S_\nu^m$	is onto
(vi)	$\beta^{n-1} < \nu^m - 1$	$\Rightarrow$ T:	$S_\beta^n \to S_\nu^m$	is not onto
(vii)	$\beta^{n-1} > \nu^m - 1$	$\Rightarrow$ R:	$S_\beta^n \to S_\nu^m$	is onto
(viii)	$\beta^{n-1} < \nu^m - 1$	$\Rightarrow$ R:	$S_\beta^n \to S_\nu^m$	is not onto

For brevity we shall prove here only parts i, ii, and vi which are illustrative of the methods used. The complete proof is available in reference (1).

(i) $\nu^{m-1} \geq \beta^n - 1 \Rightarrow T: S_\beta^n \to S_\nu^m$ is one-to-one:

Assume that $\nu^{m-1} \geq \beta^n - 1$.

Note that this is equivalent to saying that the minimum gap in S_β^n is greater than or equal to the maximum gap S_ν^m.

$$g(S_\beta^n) \geq G(S_\nu^m)$$

Given $x, y \in S_\beta^n$, $x \neq y$, we must show $T(x) \neq T(y)$. If either x or y is zero or if x and y are of opposite sign the result is immediate. Furthermore if both x and y are negative and $T(x) = T(y)$, then also $T(-x) = T(-y)$.

Hence it is sufficient to consider the case $x > y > 0$.

$$\begin{aligned} x &= y + \{(x-y)/y\}\, y \\ &\geq T(y) + g(S_\beta^n)\, T(y) \\ &\geq T(y) + G(S_\nu^m)\, T(y) \\ &\geq T(y) + \left\{\{[T(y)]' - T(y)\}/T(y)\right\} T(y) \\ &= [T(y)]' \end{aligned}$$

Thus

$$T(x) \geq [T(y)]' \text{ and } T(x) \neq T(y).$$

This proves the sufficiency of the condition $\nu^{m-1} \geq \beta^n - 1$ for the mapping T to be one-to-one.

(ii) $\nu^{m-1} < \beta^n - 1 \Rightarrow$ T: $S_\beta^n \rightarrow S_\nu^m$ is not one-to-one.

Let T: $S_\beta^n \rightarrow S^m$ and assume $\nu^{m-1} < \beta^n - 1$.

Multiplying by $(\nu^{1-m}\, \beta^{-n})$

$0 < \nu^{1-m} - \beta^{-n} - \beta^{-n}\nu^{1-m}$

or

$1 < (1-\beta^{-n})\,(1+\nu^{1-m})$

so that

$1/(1-\beta^{-n}) < (1+\nu^{1-m})$

By theorem 1 there exists integers i, j > 0, such that

$1/(1-\beta^{-n}) < \beta^i/\nu^j < 1 + \nu^{1-m}$

Thus

$\nu^j < \beta^i\,(1-\beta^{-n}) < \nu^j\,(1+\nu^{1-m})\,(1-\beta^{-n})$

$< \nu^j\,(1+\nu^{1-m})$

and using corollary 1 of the definition of successor

$$\nu^j < \beta^i\,(1-\beta^{-n}) < [\beta^i\,(1-\beta^{-n})]'$$

$$= \beta^i < \nu^j\,(1+\nu^{1-m}) = [\nu^j]'$$

The inequalities above imply $T(\beta^i\,(1-\beta^{-n})) = T\,(\beta^i)$, so that T: $S_\beta^n \rightarrow S_\nu^m$ is not one-to-one.

(vi) $\beta^{n-1} < \nu^m - 1 \Rightarrow$ T: $S_\beta^n \rightarrow S_\nu^m$ is not onto:

Given $g(S_\nu^m) < G(S_\beta^n)$, then by (ii) the truncation map of S_ν^m into S_β^n is not one-to-one, so that for some $y \in S_\nu^m$, $x \in S_\beta^n$ (x,y>0 may be assumed) where the successors are taken in the appropriate spaces.

Now if y > x, then T: $S_\beta^n \rightarrow S_\nu^m$ is not onto.

If y = x by theorem 1 we choose i,j > 0 such that

$$x'/y' > \nu^j/\beta^i > 1$$

and observing that $\beta^i\,[x]' = [\beta^i x]'$, $\nu^j\,[y]' = [\nu^j y]'$,

$$\beta^i\, x < \nu^j y < [\nu^j y]' < [\beta^i x]'$$

Hence $\nu^j y \in S_\nu^m$ is not covered and T is not onto.

Observe that the one to one and onto conditions of the Base Conversion Theorem yield immediately the following:

Corollary: The truncation (rounding) conversion map of S_β^n into S_ν^m is onto if and only if the truncation (rounding) conversion map of S_ν^m into S^n is one-to-one.

CONCLUSIONS AND APPLICATIONS

Base conversion introduces a form of error purely of numerical origin and not due to any uncertainty of measurement; the reality of this error, however, cannot be ignored. Just as it is preferable in reporting experimentally determined values to give a number of significant digits which reflect the accuracy of the measurement (no fewer digits, since useful information may be made unavailable to other researchers; no more digits, since the implication of additional precision is unfounded and may lead to spurious deductions), it is similarly desirable, after conversion of a number, to have the number of "significant" digits in the new base reflect the precision previously available with the number of digits used in the old base. Therefore, it is reasonable to require that the new base should have no fewer significant digits than the number necessary to allow the conversion mapping to be onto, nor more significant digits than is necessary for the conversion to be one-to-one. The Base Conversion Theorem has shown, in general, that these bounds on the number of digits must be different, and we now consider what the bounds are and how big are their respective differences.

For the conversion of $S^n{}_\beta$ into a space with base ν, we define both an onto number and a one to one number.

Definition: The onto number, $O(S_\beta^n,\nu)$, and one-to-one number, $I(S_\beta^n,\nu)$, for the conversion mapping of S_β^n into a space with base ν, are given by

$O(S_\beta^n, \nu) = \max\ \{m \mid T: S_\beta^n \rightarrow S_\nu^m \text{ is onto}\}$

$I(S_\beta^n, \nu) = \min\ \{m \mid T: S_\beta^n \rightarrow S_\nu^m \text{ is one-to-one}\}$

Note that R (rounding conversion) could have been used instead of T (truncation conversion) in the definition, and the same result would be obtained. From the Base Conversion Theorem we have:

Corollary 1: If $\beta^i \neq \nu^j$ for non-zero integers i, j, then

(i) $O(S_\beta^n, \nu) = \max\ \{m \mid \beta^{n-1} \geq \nu^m - 1,\ m \text{ an integer}\}$

(ii) $I(S_\beta^n, \nu) = \min\ \{m \mid \nu^{m-1} \geq \beta^n - 1,\ m \text{ an integer}\}$

(iii) $I(S_\beta^n, \nu) > O(S_\beta^n, \nu)$

For computer applications the interesting conversions are between decimal and some variant of binary. Thus for "input" conversions, let $\beta = 10$, $\nu = 2^i$, where i is 1 for binary, 3 for octal, and 4 for hexadecimal. Observe that the condition $10^{n-1} \geq 2^{im} - 1$ cannot be an equality for any $n \geq 2$, since the left hand side is then even and greater than zero whereas the right hand side is either odd or zero.

Thus for $n \geq 2$ and $10^{n-1} \geq 2^{im}$ we have

$$m \leq \frac{(n-1)}{i} \log_2 10$$

Using the number theoretic function [x] for the greatest integer in x yields

Corollary 2: For $n \geq 2$, $i \geq 1$,

$$O(S_{10}^n, 2^i) = [n(\log_2 10)/i - (\log_2 10)/i]$$

By reasoning as above with $n \geq 1$, $2^{i(m-1)} \geq 10^n - 1$ implies that $2^{i(m-1)} > 10^n$ so that:

Corollary 3: For $n \geq 1$, $i \geq 1$,

$$I(S_{10}^n, 2^i) = [n(\log_2 10)/i + 2]$$

Note that both $O(S_{10}^n, 2^i)$ and $I(S_{10}^n, 2^i)$ are, except for the fact that they must take on integral values, linear functions of n with the same slope $(\log_2 10)/i$. Thus the difference between $O(S_{10}^n, 2^i)$ and $I(S_{10}, 2^i)$ is essentially constant and does not grow with n, being equal to either $[(\log_2 10)/i] + 2$ or $[(\log_2 10)/i] + 3$.

Table I lists the cases of interest for conversion of decimal input to the commonly used variants of binary representation in computer hardware.

For completeness note that the "output" conversions, $\beta = 2^i$ and $\nu = 10$, can be derived by the above approach.

Corollary 4:

$$O(S_{2^i}^n, 10) = [n(i \log 2) - i(\log 2)] \text{ for } n \geq 1, i \geq 1$$

$$I(S_{2^i}^n, 10) = [n(i \log 2) + 2] \text{ for } n \geq 2, i \geq 1$$

The criteria for a conversion map to be one-to-one, is strictly speaking, the necessary condition for the whole space S^n to be mapped into S^m in a

	BINARY		OCTAL		HEXADECIMAL	
NUMBER OF DECIMAL DIGITS	Maximum Digits for Onto Conversion	Minimum Digits for One-to-One Conversion	Maximum Digits for Onto Conversion	Minimum Digits for One-to-One Conversion	Maximum Digits for Onto Conversion	Minimum Digits for One-to-One Conversion
1	1	5	0	3	0	2
2	3	8	1	4	0	3
3	6	11	2	5	1	4
4	9	15	3	6	2	5
5	13	18	4	7	3	6
6	16	21	5	8	4	6
7	19	25	6	9	4	7
8	23	28	7	10	5	8
9	26	31	8	11	6	9
10	29	35	9	13	7	10
11	33	38	11	14	8	11
12	36	41	12	15	9	11
13	39	45	13	16	9	12
14	43	48	14	17	10	13
15	46	51	15	18	11	14
16	49	55	16	19	12	15
17	53	58	17	20	13	16
18	56	61	18	21	14	16
19	59	65	19	23	14	17
20	63	68	21	24	15	18
21	66	71	22	25	16	19
22	69	75	23	26	17	20
23	73	78	24	27	18	21
24	76	81	25	28	19	21
25	79	85	26	29	19	22

TABLE I: Bounds for Onto and One to One Conversions of Decimal to Binary, Octal, or Hexadecimal Systems

one-to-one fashion. Naturally certain ranges of values in S^n may be mapped in a one-to-one fashion even though the criteria $\nu^{m-1} \geq \beta^n - 1$ is not satisfied.

The computer hardware representation of a number places a limit on the size of the exponent as well as on the number of digits allowed. Thus only a given range of numbers in S^n_β, say, e.g., between 10^{-40} and 10^{40}, need be mapped one-to-one in order to effect a one-to-one conversion. However, unless the decision criteria for a one-to-one conversion, $\nu^{m-1} \geq \beta^n - 1$, is very close to being satisfied, there will almost certainly be a value x well within the range $10^{-40} \leq x \leq 10^{40}$ such that x and x′ are converted to the same number. For example, it may be verified that for decimal to hexadecimal conversion the number .0625 and its successor in S^n_{10} will both be truncated to the same value in S^I_{16} (S^n_{10},16)−1 for all n from 3 to 25 except n=5, 11, 17, 23. With decimal to octal conversion, 8 and its successor in S^n_{10} will both be truncated to the same value in S_8, (j=I(S_{10}, 8)−1), for n from 3 to 25 except n=10, 19, and an even more critical region for decimal to octal conversion is the neighborhood of .991 x 10^{28}. Our point is that the necessary bounds reported in Table I are not required because of pecularities with pathologically large or small numbers, but are generally applicable for the range of magnitude of floating point numbers handled in computer hardware.

For the computer programmer utilizing decimal input, a one-to-one conversion into machine representation is desirable from the point of view of maintaining distinctness between two unequal decimal numbers. On the other hand an onto conversion map allows the programmer to achieve every distinct numerical machine word and, therefore, more fully utilize the machine hardware capabilities. Unfortunately decimal-binary conversion cannot be both one-to-one and onto, so some compromise must be made.

There is one case, in this author's opinion, where the decision in favor of a one-to-one conversion should be mandatory. That is in a more sophisticated implementation of a user language where the number of digits in a decimal constant is used to signal either "single precision" or "double precision" machine representation. Then the allowable constants which specify single precision should be in a range where conversion into machine representation is by a one-to-one mapping. This is necessary to prevent the anomally of having two distinct decimal constants become identical in the single precision machine representation when these same two decimal constants written with trailing zeros (forcing double precision representation) would receive distinct machine representations in double precision.

For example in an implementation of FORTRAN where normalized decimal constants of 7 or less digits are converted by truncation (or by rounding for that matter) to 6 hexadecimal digit normalized numbers and where 8-16 digit decimal constants are converted to 14 hexadecimal digit normalized numbers, the logical expression

512.0625 .EQ. 512.0626

would be true whereas

512.06250 .EQ. 512.06260

would be false. This problem could and should be eliminated by specifying, according to our theorem, single precision only for constants of 6 or less decimal digits, and double precision for constants of 7 or more decimal digits. Alternatively a more refined test could be made by first determining if the number $y \in S^7_{10}$ falls in a range where $\gamma^6_{16}(y) > \gamma^7_{10}(y)$, and if so, then prescribe double precision representation for y.

Figure 2 shows that those intervals where $\gamma^6_{16}(y)$ exceeds $\gamma^7_{10}(y)$ are erratically spaced and cover a substantial portion, amounting to about 40% on the

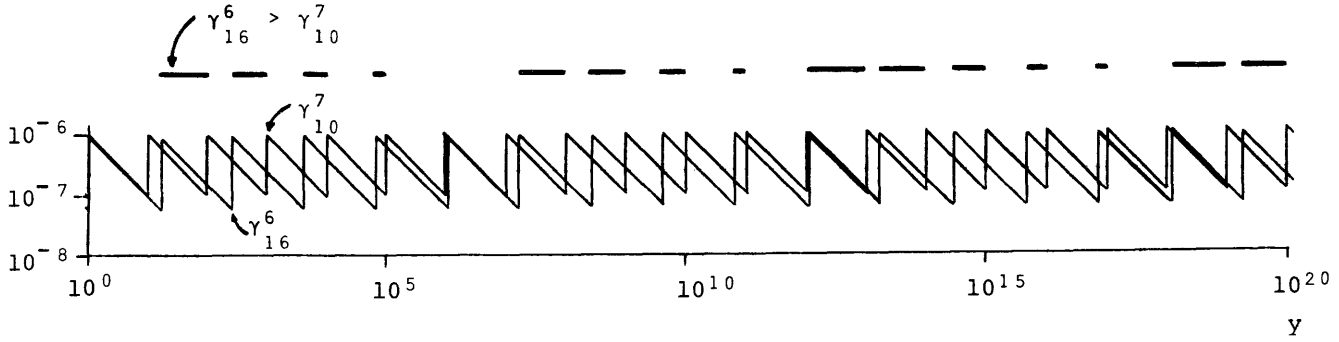


Figure 2—Comparison of the gap functions $\gamma^7_{10}(y)$ and $\gamma^6_{16}(y)$. The subintervals where $\gamma^6_{16} > \gamma^7_{10}$ are shown to comprise about 40% of the interval $1 \leq y \leq 10^{20}$ (measured on the log scale)

log scale, of the interval $1 \leq y \leq 10^{20}$. This agrees very well with the limiting frequency of the range where $\gamma_{16}^{6}(y)$ exceeds $\gamma_{10}^{7}(y)$, which is given by $[\log(16^{-5}/10^{-7})]/(2 \log 16) = 40.68\%$.

Utilizing the comparison of gap functions we can verify that a simple test allowing a one-to-one conversion mapping of S^{7}_{10} into either S^{6}_{16} or S^{14}_{16} is to map a given number of S^{7}_{10} into S^{6}_{16} if the leading non-zero hexadecimal digit of the converted number is strictly greater than the leading non-zero digit of the original decimal number, and otherwise map into S^{14}_{16}.

In general, the practical relevance of the Base Conversion Theorem to digital computer data handling can be summarized by observing that if we were to postulate the goals of Table II there is an essential incompatibility between the uniqueness of conversion goal and the optimal utilization of storage goal. Thus when a "systems" decision on the number of digits allowable for decimal constants in the implementation of a user language is made, the user should become aware of where and why the compromise was made to avoid certain "unexpected" results.

Goals of Numerical Representation	Goals of Machine Design
(1) Accuracy and Uniqueness of conversion	(1) Optimal use of available storage
(2) Accuracy of calculations	(2) Convenience of machine logic

Table II

REFERENCES

1 D W MATULA
Base Conversion Mappings
Report No AM-66-1 School of Engineering and Applied Science Washington University St Louis Mo

Accurate solution of linear algebraic systems–a survey

by CLEVE B. MOLER
University of Michigan
Ann Arbor, Michigan

INTRODUCTION

Many problems encountered in computing involve the solution of the simultaneous linear equations

$$A\,x = b \tag{1}$$

where A is a n-by-n matrix,

$$A = \begin{pmatrix} a_{1,1} & \circ & \circ & \circ & a_{1,n} \\ \circ & & & & \circ \\ \circ & & & & \circ \\ \circ & & & & \circ \\ a_{n,1} & \circ & \circ & \circ & a_{n,n} \end{pmatrix}$$

and b and x are vectors.

Most people interested in computing are familiar with the basic concepts involved in solving such a system, but there are several useful refinements and extensions that are not so well known. It is the purpose of this article to introduce the non-expert to these ideas.

Several of the newer ideas, as well as clarifications of older ones, are due to J. H. Wilkinson and are summarized in his book.[1] Many people, including this writer, are indebted to G. E. Forsythe for their knowledge of this field. Forsythe's recent survey[2] contains an extensive bibliography together with material on related topics. A forthcoming text[3] contains most of the details we will omit here, as well as computer programs which implement the techniques.

Gaussian elimination and triangular decomposition

The most common method for solving (1) is Gaussian elimination. Unless it is known that A has some special properties, such as being positive definite or having many zero elements, this method is also the most efficient in the sense that no other can involve fewer basic arithmetic operations. With this method, it is easy to solve a 150-by-150 system in a few minutes on many computers. Larger systems often involve special matrices that make other methods more appropriate.

As it is usually practiced, Gaussian elimination begins by attaching the right hand side b to the matrix A as an additional column, $a_{i,n+1} = b_i$, $i = 1, \ldots, n$. Let us denote this initial rectangular array by $a^{(1)}_{i,j}$. Then the *forward elimination* is performed by subtracting appropriate multiples of each row from the succeeding rows. The algorithm is

$$\begin{aligned} &\text{for } k = 1, \ldots, n-1, \\ &\quad \text{for } i = k+1, \ldots, n, \\ &\qquad m_{i,k} = a^{(k)}_{i,k}/a^{(k)}_{k,k} \\ &\quad \text{for } j = k+1, \ldots, n+1 \\ &\qquad a^{(k+1)}_{i,j} = a^{(k)}_{i,j} - m_{i,k} \circ a^{(k)}_{k,j} \circ \end{aligned} \tag{2}$$

This produces the triangular array

$$\begin{matrix} a^{(1)}_{1,1} & a^{(1)}_{1,2} & a^{(1)}_{1,3} & \cdots & a^{(1)}_{1,n} & a^{(1)}_{1,n+1} \\ & a^{(2)}_{2,2} & a^{(2)}_{2,3} & \cdots & a^{(2)}_{2,n} & a^{(2)}_{2,n+1} \\ & & \bullet & \cdots & \bullet & \bullet \\ & & & & a^{(n)}_{n,n} & a^{(n)}_{n,n+1} \end{matrix}$$

Letting $y_i = a^{(i)}_{i,n+1}$ $i = 1, \ldots n$, this array represents a set of equations

$$A^{(n)}\,x = y$$

that is equivalent to the original one, but which is easily solved. The *back substitution* carries out this solution by using

$$\text{for } i = n, n-1, \ldots, 1,$$

$$x_i = [y_i - \sum_{j=i+1}^{n} a^{(i)}_{i,j}\, x_j]/a^{(i)}_{i,i} \tag{3}$$

The numbers $a^{(k)}_{k,k}$ are called the *pivots*. If any of them are zero, the corresponding division can not be done and the above algorithms break down. We will return to this point. The quantities $m_{i,k}$ are called the *multipliers*.

Some applications require solution of equations involving the same matrix A, but several different right hand sides b, each of which depends upon the solution corresponding to a previous right hand side. To handle this situation, we observe the matrix $A^{(n)}$ need be

computed only once and used with as many right hand sides as necessary. The algorithm for computing $A^{(n)}$ is (2), with j going only to n instead of n+1. This requires roughly $\frac{1}{3}n^3$ multiplicative operations.

Once $A^{(n)}$ and the multipliers $m_{i,k}$ are available, a right hand side can be processed with two simple steps requiring only n^2 multiplicative operations. The first step is the $j = n + 1$ part of (2) which can be rewritten

$$\text{for } i = 1, 2, \ldots, n,$$
$$y_i = b_i - \sum_{j=1}^{i-1} m_{i,j}\, y_j \,. \tag{4}$$

The second step is the back substitution (3).

This interpretation of Gaussian elimination is also important for other reasons. It may be summarized by using matrix, rather than component, notation. Let

$$L = \begin{pmatrix} 1 & & & & \\ m_{2,1} & 1 & & & \\ m_{3,1} & m_{3,2} & 1 & & \\ \cdot & \cdot & \cdot & \cdot & \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & m_{n,n-1}\ 1 \end{pmatrix}$$

and

$$U = A^{(n)} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ & a^{(2)}_{2,2} & a^{(2)}_{2,3} & \cdots & a^{(2)}_{2,n} \\ & & a^{(3)}_{3,3} & \cdots & a^{(3)}_{3,n} \\ & & & \cdots & a^{(n)}_{n,n} \end{pmatrix}$$

Then it is not hard to show that

$$A = L\,U\,.$$

The matrices L and U together form a *triangular* or *LU-decomposition* of A. Note that the diagonal elements of U are the pivots.

Since $A = LU$, the system of equations $Ax = b$ becomes $LUx = b$ which is actually the two easily solved triangular systems $Ly = b$ and $Ux = y$. In particular, this accounts for the similarity between (4) and (3).

Several right hand sides could also be handled by first computing A^{-1}, but this is less efficient and usually less accurate than using L and U.

Pivoting and scaling

We observed that (2) and (3) are impossible if any of the pivots are zero. Computing intuition consequently tells us that (2) and (3) may involve serious roundoff errors if any of the pivots are nearly zero. This can be confirmed by example. The solution of the system

$$\begin{pmatrix} 10^{-4} & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix},$$

rounded to 3 significant digits, is

$$x_1 = 1.00,\ x_2 = 1.00.$$

Naive application of (2) on a computer carrying 3 significant digits gives

$$\begin{pmatrix} 10^{-4} & 1 \\ 0 & -10^{4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -10^{4} \end{pmatrix}.$$

Following with (3), we obtain

$$x_1 = 0,\ x_2 = 1.00$$

which is a terrible result.

On the other hand, if we first interchange the two equations above, we obtain the correctly rounded solution without difficulty.

A key to the trouble is the fact that without the interchange the multiplier $m_{2,1}$ is 10^4, whereas with the interchange it is 10^{-4}. It turns out in general that if the multipliers satisfy

$$|m_{i,k}| \leq 1, \tag{5}$$

then the computed solution can be proved to be accurate.

Condition (5) can be insured by the use of *partial pivoting*: At the k-th step of the forward elimination, the pivot is taken to be the largest (in absolute value) element in the unreduced part of the k-th column, that is $\{a^{(k)}_{i,k}, i = k, \ldots, n\}$. This can be implemented on a computer by either actually interchanging the appropriate rows in storage or by using a double indexing scheme involving references to $a_{p(i),j}$ in place of $a_{i,j}$. The latter choice appears more efficient in most situations. For details, see reference 3.

The use of *complete pivoting* involves finding the largest element in the entire unreduced submatrix. It has some rare advantages ([1], p. 97), but involves much more computer time. Experience indicates that partial pivoting is quite satisfactory in practice.

If a row or column of A is multiplied by some scale factor, it is easy to make a compensating change in the computed x. Such scale factors are usually considered arbitrary in floating-point computation. However, they are intimately connected with pivoting and hence with accuracy. A common practice is to *equilibrate* the matrix beforehand so that the maximum element in each row and column is between 1 and 10, say.

However, consider the matrix

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 10^{9} & -1 & 1 \\ 10^{9} & 1 & 0 \end{pmatrix}$$

A column scaling yields

$$A_c = \begin{pmatrix} 10^{-9} & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

whereas a row scaling yields

$$A_r = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -10^{-9} & 10^{-9} \\ 1 & 10^{-9} & 0 \end{pmatrix}.$$

While both A_c and A_r are equilibrated, A_c will be well behaved in any pivoting routine, but A_r will produce a singular submatrix.

This shows that a matrix may have several equilibrated forms, some of which lead to excessive roundoff errors. The problem of finding better scaling criteria

is, as yet, unsolved. Fortunately, seriously scaling difficulties do not appear to be too common.

Accuracy of the computed solution

Let x_1 denote the vector actually computed using (2), (4) and (3). Wilkinson[1] has shown that there is a matrix dA and so that x_1 exactly satisfies the equation

$$(A + dA)\, x_1 = b. \tag{6}$$

In other words, the computed solution x_1 is the exact solution to a slightly different original equation. The matrix dA depends upon b and can be expressed, and actually computed if desired, in terms of the individual rounding errors at each step of the algorithm.

It is possible to give an upper bound for the size of dA which does not involve b or the details of each step. In order to state the result, we define the norm of a vector to be

$$||z|| = \max_i |z_i|$$

and the corresponding norm of a matrix to be

$$||A|| = \max_{||z|| = 1} ||Az||$$

$$= \max_i \sum_{j=1}^{n} |a_{i,j}| \ .$$

Furthermore, let ϵ denote the roundoff (or chopoff) level of the arithmetic being used. This is the largest number for which

$$1.0 + \epsilon = 1.0.$$

Finally, let

$$\rho = \max_{i,j,k} \frac{|a^{(k)}_{i,j}|}{||A||}$$

This is the size (relative to the original A) of the elements of the reduced submatrices. It can be computed during the elimination and is almost always less than one.

With these definitions and Wilkinson's methods, it can be shown that

$$\frac{||dA||}{||A||} \leq (n^3 + 3n^2)\ \rho\ \epsilon\ . \tag{7}$$

In practice the quantity $||dA||/||A||$ is found to be considerably less than the indicated bound.

It is easy to use (6) and (7) to derive a bound for the error in x_1. If x is the true solution to (1), then

$$\frac{||x_1 - x||}{||x_1||} \leq ||A|| \cdot ||A^{-1}|| \cdot (n^3 + 3n^2)\ \rho\ \epsilon \tag{8}$$

The quantity $||A|| \cdot ||A^{-1}||$ is the *condition* of A. It is a useful measure of the "nearness to singularity" of A.

Iterative improvement

If A and b are themselves subject to error of some kind, then (6) and (7) indicate that x_1 may be as accurate an answer as is justified, in the sense that it is the exact answer to some system within the range of uncertainity of the given system.

Frequently however, the components of A and b are assumed to be exact floating point numbers, and it is desired to compute the solution as accurately as possible. In this case, the error in x_1 may be reduced by an iterative process. For $m = 1, 2, \ldots$, the m*th* iteration involves three steps:

(S1) Compute the residuals, $r_m = b - Ax_m$,
(S2) Solve the system, $Ad_m = r_m$,
(S3) Add the correction, $x_{m+1} = x_m + d_m$.

The LU-decomposition of A used in finding the first approximation x_1 may be used to solve the subsequent systems involved in (S2). Consequently, the complete iterative process usually requires only a fraction of the time originally required to compute x_1.

The critical portion of the process is (S1). It is necessary here to use an *extended accumulated inner product* which calculates the quantity

$$b_i - \sum_{j=1}^{n} a_{i,j}x_j$$

using multiplications which produce a high precision product from working precision numbers and additions which retain the high precision. The final result is truncated to working precision for subsequent use. We say "working" and "high" rather than "single" and "double" because "working" may already be "double."

For floating-point arithmetic, convergence of the process and the required relationship between high and working precision are investigated.[4] This is an extension of the fixed-point analysis in [1].

To see the typical behavior of the process, let us say that for a particular matrix A and any right hand side, Gaussian elimination produces an answer with s correct significant figures. Then x_1 will have s correct figures, d_1 will also have s correct figures, hence x^2 will have 2s correct figures, x_3 will have 3s, and so on. The convergence is linear at a rate determined by the accuracy of (S2).

If $s = 0$ then none of the x_m's will have any accuracy and the complete Gaussian elimination would have to be done using higher precision.

To be somewhat more precise, let $x = A^{-1}b$ be the true solution, and let $k(A) = ||A|| \cdot ||A^{-1}||$ be the condition of A and let ϵ_2 be the roundoff level of high precision addition. Then, ignoring less important terms,

$$\frac{||x_m - x||}{||x||} \leq [n^3 \rho\, k(A)\, \epsilon]^m + \epsilon + nk(A)\epsilon_2. \tag{9}$$

The three terms on the right come from (S2), (S3) and (S1) respectively. A sufficient condition for convergence is that the quantity in brackets is less than 1, i.e. that A is not too badly conditioned. If this is satisfied, the error decreases until a limiting accuracy determined by the larger of the other two terms is reached. If it is also true that $\epsilon_2 \leq \epsilon^2$, i.e. that high precision is at least twice working precision, then the limiting accuracy is ϵ.

Neither of these conditions is very restrictive and in almost all practical cases, x_3 or x_4 is found to be equal to the true solution, rounded to working precision.

An example

We conclude with a 2-by-2 example:

$$A = \begin{pmatrix} 1.0303 & .99030 \\ .99030 & .95285 \end{pmatrix}, \quad b = \begin{pmatrix} 2.4944 \\ 2.3988 \end{pmatrix}.$$

Using 5-digit, decimal floating-point arithmetic, we compute (no pivoting is necessary)

$$L = \begin{pmatrix} 1 & 0 \\ .96118 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1.0303 & .99030 \\ 0 & .00099 \end{pmatrix}$$

and

$$x_1 = \begin{pmatrix} 1.2560 \\ 1.2121 \end{pmatrix}.$$

With a 10-digit accumulated inner product, we find

$$r_1 = \begin{pmatrix} .57000 \cdot 10^{-6} \\ .33715 \cdot 10^{-4} \end{pmatrix},$$

and then, using L and U,

$$d_1 = \begin{pmatrix} -.32210 \cdot 10^{-1} \\ .33502 \cdot 10^{-1} \end{pmatrix}.$$

Continuing,

$$x_2 = \begin{pmatrix} 1.2238 \\ 1.2456 \end{pmatrix}, \quad r_2 = \begin{pmatrix} .188 \cdot 10^{-5} \\ .900 \cdot 10^{-6} \end{pmatrix}, \quad d_2 = \begin{pmatrix} .2285 \cdot 10^{-3} \\ -.2365 \cdot 10^{-3} \end{pmatrix},$$

$$x_3 = \begin{pmatrix} 1.2240 \\ 1.2454 \end{pmatrix}, \quad r_3 = \begin{pmatrix} -.682 \cdot 10^{-5} \\ -.659 \cdot 10^{-5} \end{pmatrix}, \quad d_3 = \begin{pmatrix} .2717 \cdot 10^{-4} \\ -.3315 \cdot 10^{-4} \end{pmatrix},$$

$x_4 = x_3$; stop.

It turns out that $x_3 = A^{-1}b$ to the 5 figures given. Note that x_1 is accurate to only 2 figures. Since we used 5-figure arithmetic, this indicates that k(A) must be roughly 10^3. In fact, $k(A) = 3.97 \cdot 10^3$. Note also that r_3 is actually larger than r_2, even though the error in x_3 is less than that in x_2. This phenomenon is typical, even in larger systems.

REFERENCES

1 J. H. WILKINSON
Rounding errors in algebraic processes
H. M. Stat. Office and Prentice Hall 1963

2 GEORGE E. FORSYTHE
Today's computational methods of linear algebra
to appear in SIAM Review

3 GEORGE E. FORSYTHE and CLEVE B. MOLER
Computer solution of linear algebraic systems
Prentice Hall to appear

4 CLEVE B. MOLER
Iterative refinement in floating point
J. Assoc. Comput. Mach. vol. 14 1967

Recursive techniques in problem solving

by A. JAY GOLDSTEIN
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

INTRODUCTION

Recursive methods have long been a standard tool of compiler writers. However, other kinds of computer users have not fully realized that recursion is a powerful technique for solving and programming problems.* Even when a language with recursive facilities is not available, an attack on a problem from a recursive point of view will often show how an iterative program can be written.

In this paper, three problems are approached using recursion: generation of permutations of N objects, evaluation of polynomials in several variables, and generation of all the spanning trees of a graph. These examples illustrate that, when applicable, recursive methods significantly reduce the difficulties of (1) formulating a solution algorithm for the problem, (2) programming the algorithm, and (3) making the program readable.

For those not familiar with recursive programming, the following anthropomorphic analogy may be instructive. John wishes to use a recursive program P which has within it a call to P. He makes a large number of printed copies of the program P and obtains the services of a large number of subordinates. To use the program P, John chooses a subordinate John 1, gives him a copy of P with a list of the arguments of P and tells him to follow the instructions in P and to report back when he is finished with his task. John 1 starts his task and finds in the process that he must call for program P with a new list of arguments. John 1 then chooses a subordinate, John 2, gives him a copy of P with the new list of arguments, and tells him to follow the instructions in P and to report back when he is finished with his task. John 2 starts his task and may have to choose a subordinate, John 3, who may have to choose a subordinate, John 4, etc. Eventually, a subordinate John N starts his task and finishes it without needing a subordinate. John N reports his results to his supervisor John N-1, who completes his task and reports his results to his supervisor John N-2, etc. Sometimes an intermediate supervisor, after receiving the result from his subordinate, may have to choose another subordinate in order to make a further call to program P. Thus, the chain of supervisors can oscillate in length before John 1 finally completes his task and reports the results to John. John is happy.

The actual way in which a language with recursive facilities is implemented on a computer need not concern us here. However, the analogy gives the essence of how recursion works.

Permutations

The need to generate all the arrangements of N distinct objects often arises in combinatoric problems. Here is one such problem.

In the layout of electrical packages, one often wants to place the packages on a frame so as to minimize the amount of wire needed to make the specified connections between packages. Sometimes the actual cost of wiring is crucial, but more often the desire is to minimize electromagnetic and electrostatic coupling between wires by minimizing the amount of wire. Consider the simple one-dimensional problem where the packages are all identical in shape and must be placed along a line. Let us simplify the problem further by assuming that the packages (N+2 of them) are points and are to be placed at the integer points 0, 1,...,N+1. The designer insists that two of the packages, the 0-th and (N+1)st which are input-output packages, must be located at 0 and N+1. The interconnections are specified by C(I,J) – the number of connections between packages I and J. The location of package I is denoted by P(I) with P(0)=0 and P(N+1) = N + 1.

A naive approach to the minimization problem would be to compute the amount of wire

$$\sum_{I<J} C(I,J) \mid P(I) - P(J) \mid$$

*Dynamic programming is one special form of recursion in use, but its area of applicability is limited.

for each of the N! arrangements. This requires N(N−1)/2 subtractions and multiplications for each arrangement.

A more sophisticated approach would be to generate the arrangements in such a way that the incremental change in wire length, from one arrangement to the next, would be easy to compute. One way to accomplish this is to generate the arrangements so that each differed from its predecessor by an interchange of adjacent packages. Then the incremental change in wire length can easily be computed by keeping tables NL(J) and NR(J) of the number of wires from package J to packages on its left and right, respectively. If the next arrangement is obtained by interchanging packages L and R in positions p and p + 1, then the incremental change in wire length is

$$[NL(L) - (NR(L) - C(L,R))] + [NR(R) - (NL(R) - C(L,R))]$$

Due to the interchange, the quantities NL(J), NR(J) for J = R,L must be modified by C(L,R).

Using this method of generating arrangements, the amount of computation per arrangement is very small and is independent of N, in contrast to the original method in which the amount of computation is proportional to N^2.

There is a tacit assumption that the importance of obtaining an absolute minimum is great enough to warrant an exhaustive search through the N! arrangements. If the importance is not that great, then successive approximation techniques are available.

We now ask the crucial question: Can all the arrangements on N distinct objects be generated one after another such that each differs from its predecessor by an interchange of adjacent objects?

A recursive approach to this problem begins with the hypothesis: suppose we know how to do it for N − 1 objects. A little thought and inspiration will result in the following. From each arrangement a . . . f on N − 1 objects we can generate N permutations ga . . . f, ag . . . f, . . . , a . . . gf, a . . . fg on N objects where each differs from the preceding by an interchange of adjacent objects. Applying the hypothesis, we take the next arrangement on N − 1 objects a′ . . . f′ (differing from a . . . f by an adjacent interchange) and append g on the right. Then, by repeatedly interchanging g with its left neighbor, generate the next N arrangements. This elegant recursive algorithm is due to S. M. Johnson.[1]

Frequently we want a program which upon each call returns the next permutation. INTER, given below, is such a function procedure written in PL/I. Each CALL returns an integer INTER which says: "to obtain the next permutation, interchange the objects in position INTER and INTER+1". INTER = 0 indicates that all permutations have been generated. Note that the user starts with one arrangement and hence needs only N! − 1 calls to INTER. The N! call must be made − with a return of INTER = 0 − in order to clean up the program.

The previous discussion of the generation process shows that INTER

(1,N) − must know location of the N-th object a_N with respect to $a_1, \ldots, a_N$,

(2,N) − must know whether a_N is shifting right or left, and

(3,N) − must take special action if a_N is at the left or right end. I.E. the direction of shift must be reversed and a recursive call to INTER must be made to find out what the next interchange on $a_1, \ldots, a_{N-1}$ is.

At the next level of recursion produced by the call (3,N), the procedure must have the information in (1,N−1) and (2,N−1) and take the action in (3,N−1). In general (1,K), (2,K) and (3,K) are needed.

To store this information INTER uses a one-dimensional array P of length N. The sign of P(K) is plus or minus if the K-th object is moving right or left, respectively. The magnitude of P(K) is the relative position of the K-th object, that is, | P(K) | is 1 + the number of objects $a_1, \ldots, a_{K-1}$ to the left of the a_K.

INTER must handle three cases. Case 1. P(K) = K, i.e., a_K is at the right end and shifting right. Then the direction of shift must be reversed by replacing P(K) by −K and the position INTER(K−1) returned. Recall that this says that the next interchange involves the objects in position INTER(K−1) and INTER(K−1) + 1. Case 2. P(K) = −1, i.e., a_K is at the left end and is shifting left. Then the direction of shift must be reversed by replacing P(K) by +1 and the position INTER(K−1) + 1 returned − one is added because a_K is at the left. Case 3. P(K) ≠ K,−1. In this case P(K) is returned if P(K) is positive and |P(K)|−1 is returned if P(K) is negative, and P(K) is updated to P(K) + 1.

```
INTER: PROCEDURE (K) FIXED RECUR-
       SIVE;
       DECLARE K FIXED, P(50) FIXED
       STATIC;
       /*ARE WE FINISHED*/
       IF K ≤ 1 THEN
       DO; P = 1; RETURN
       (0); END;
       /*CASE 1*/
       IF P(K) = K THEN
       DO; P(K) = −K; RETURN (INTER
```

```
(K−1)); END;
/*CASE 2*/
IF P(K) = −1 THEN
DO; P(K) = 1; RETURN (INTER(K−1)
+1); END;
/*CASE 3*/
(PK) = PK + 1
IF P(K) > 0 THEN RETURN (P(K) − 1
ELSE RETURN |P)K)|);
END INTER;
```

Polynomial evaluation

Horner's method for evaluation polynomials:

$$P=A(K)=A(K\text{-}1)*X+\ldots+A(1)*X**(K\text{-}1)$$
$$=\ldots(A(3)+(A(2)+A(1)*X)*X)*X\ldots$$

has the merit of requiring only 2(K−1) arithmetic operations. In this section we show how the computational advantage of Horner's method can be extended to polynomials in several variables. First, Horner's method is examined from a recursive point of view. Second, consideration is given the storage limitations due to the astronomical number of possible monomials in a polynomial in several variables. Clearly, we can store only the nonzero monomials. The effect of this mode of storage on the previous recursive method is then examined for the special case of polynomials in one variable and a new recursive method is given. Third, this recursive method is generalized to the case of many variables.

Looking at polynomial evaluation from a recursive point of view, we note that

$$P = A(K) + X*(A(K-1) + \ldots + A(1)*X**(K-2)).$$

That is, we can evaluate a polynomial with K terms if we can first evaluate a polynomial with K − 1 terms. This is the recursive step, and we can define a recursive program

```
POLYVAL(A,X,K)=0                                  for K≤0
              =A(K) + X* POLYVAL(A,X,K−1) for K≥1.
```

POLYVAL is just Horner's method.

Before generalizing this to the several variable case, we should consider a storage problem. A polynomial in several variables can have $(d_1+1)\ (d_2+1)\ldots$ monomials where the i-th variable appears with maximal degree d_i. We cannot afford the storage for all these monomials and will store only the nonzero ones. The effect of this on Horner's method will be examined next, since, as is typical with recursive approaches, the special case is easily generalized.

Eliminating the nonzero monomials gives the polynomial

$$A(K)*X**E(K) + \ldots + A(1)*X**E(1)$$

with E(1) > E(2) > .... It can be expressed in the recursive form

$$X**E(K)*(A(K) + (A(K-1)*X**(E(K-1)-E(K)) + \ldots)).$$

Thus, we can write the recursive procedure

```
POLYVAL (A,E,X,K) = 0
                  = X**E(K)*(A(K)+
                                                  if K≤0
    POLYVAL (A,E−E(K),X,K−1)) if K≥1
```

Note that E − E(K) says: subtract E(K) from every element of E.

The PL/I procedure below is an implementation of the recursion. In order that the array E of exponents not be modified and to avoid creating a temporary array, the procedure uses the device of saving E(K) in ELAST.

```
POLYVAL:PROCEDURE   (A,E,X,K)   FOAT
        RECURSIVE;
        DECLARE(K,E(K),ELAST)
        FIXED,A(K),X) FLOAT;
        1*ELAST IS THE EXPONENT OF
        THE PREVIOUS TERM. WE CAN-
        NOT USE E(K+1) SINCE IT DOES
        NOT EXIST ON THE FIRST IN-
        VOCATION   OF   POLYVAL.*/
        ELAST = 0; GO TO START;
  P.V:  ENTRY (A,E,X,K,ELAST)FLOAT;
START:  IF K = 0 THEN RETURN (0);
        IF E(K) = 0 THEN Y = 1.0; ELSE
        Y=X**(EK)−ELAST); RETURN
        (Y*(A(K) + PV(A,E,K−1,E(K)))
        END POLYVAL;
```

To generalize POLYVAL to several variables, we must change A(K) to a polynomial in several variables and modify the RETURN statement where A(K) appears. We assume that the polynomial in VAR variables is expressed as a sum of polynomials (called terms) each consisting of the product of a power of X(VAR) and a coefficient polynomial in the variables X(1), ..., X(VAR−1). The coefficient polynomial has the same format.

Now we must specify the data structure for representing the polynomials. A PL/I structure for each

of the terms of the polynomial will work very well. The structure must contain the exponent E; a pointer, COEF, to the polynomial which is the coefficient; a pointer, NEXT, to the next term in the polynomial; and a number VALUE which is the numerical value of the coefficient when the polynomial is a constant. We assume the exponent E in the NEXT term is larger than the present exponent. With this data format decided upon, the generalization of the previous procedure to the several variable case is not difficult.

```
POLYVAL: PROCEDURE (P,X,VAR) FLOAT
         RECURSIVE;
         DECLARE
           P POINTER, /*A POINTER TO
             THE FIRST TERM IN THE
             POLYNOMIAL*/
           VAR FIXED, /*THE NUMBER
             OF VARIABLES*/
           X(VAR) FLOAT, /*THE
             VALUES OF THE VARI-
             ABLES*/
           ELAST FIXED, /*THE EX-
             PONENT OF THE PREVIOUS
             TERM*/
           1 TERM CONTROLLED (P)
             2 E FIXED, /*EXPONENT OF
               X(VAR) IN THIS TERM*/
             2 NEXT POINTER, /*POINT-
               ER TO THE NEXT TERM.
               ITS EXPONENT IS
               LARGER*/
             2 COEF POINTER, /*A
               POINTER TO THE FIRST
               TERM OF THE COEFFI-
               CIENT*/
             2 VALUE FLOAT; /*VALUE
               OF COEFFICIENT IF VAR
               = 0*/

         ELAST = 0; GO TO START;
      PV: ENTRY (P, X, VAR, ELAST)
         FLOAT;
   START: IF P = NULL THEN RETURN
         (0);
         IF VAR = 0 THEN RETURN
         (VALUE);
         IF E = 0 THEN Y = 1.; ELSE Y
         = X(VAR)**(E-ELAST);
         RETURN (Y*(PV(COEF, X, VAR
         -1, 0) + PV(NEXT, X, VAR, E));
         END POLYVAL;
```

The McIlroy method for the generation of all spanning trees

For the analysis of electrical networks by topological methods, it is sometimes necessary to find all the spanning trees of a graph. M. Douglas McIlroy has developed an elegant recursive procedure for generating them [2].

In order to understand the algorithm we will first examine one that is less efficient but more transparent. The procedure is TREE1 (G,DE,T) where G is the graph, DE a set of edges deleted from G, and T is a tree of G which does not necessarily span G. For any T, the procedure *grows* all the spanning trees which contain T as subtree. Each time a spanning tree is generated, it is handed to the user by calling his procedure JOB(T). JOB returns to the TREE1 upon completion of its task and the next tree is generated. The process is initiated by calling TREE1 with DE and T empty. We describe TREE1 in a dialect of PL/I.

```
TREE1: Procedure (G,DE,T) Recursive;
Step 1. If T is empty then T = any node of G;
Step 2. If T spans G then (call JOB(T); Return);
Step 3. Choose an edge e = (b,c) in G - DE with
        node b in T and node c not in T; If there are
        none, then return;
        /*We now grow all trees which contain T.
        These trees are divided into two disjoint
        kinds, namely, those which do not contain
        the edge e and those which do contain the
        edge e.*/
Step 4. Call TREE1 (G,DE+e,T)
Step 5. Call TREE1 (G,DE,T+e)
        Return;
        End TREE1;
```

McIlroy's algorithm TREE is obtained from TREE1 by making a crucial observation about step 3. At this step, an edge e, from node b in T to node c not in T, is chosen. At various other stages in the recursion, all the edges from c to other nodes in T are considered, one by one. This set of edges is called an attachment set. McIlroy handles all these edges simultaneously by formalizing the notion of the attachment set of a node and introducing the notion of a family of trees. The attachment set ATCH(I) of a node I is a set of nodes which are adjacent to I and describes a set of edges at node I. A family of trees is described by the array of attachment sets ATCH(1), ATCH(2), etc., one for each node of the graph G. A tree of the family is obtained by selecting for each I, one node N_I from ATCH(I) and constructing the tree whose edges are $\{(I,N_I)\}$. Each possible set of selections gives a tree.

As an example, consider the graph which is a square plus one diagonal. Its edges are (1,2) (2,3)

(3,4) (4,1) (2,4). Then one family of trees is given by

ATCH(1) = empty, ATCH(2) = {1},
ATCH(3) = {2,4}, ATCH(4) = {1,2}.

This describes the family of trees

1. (2,1) (3,2) (4,1)
2. (2,1) (3,2) (4,2)
3. (2,1) (3,4) (4,1)
4. (2,1) (3,4) (4,2)

The technique of generating the trees in families produces a tremendous increase in efficiency.

To facilitate the description of TREE, we use set notation. Let the graph G be described by giving for each node I, the *set* G(I), of nodes adjacent to I. Let NODES be the *set* of nodes of G and let TNODES be the *set* of nodes in the growing tree. Finally, let ATCH(I) be the attachment *set* at node I as above. The steps of TREE parallel those of TREE1 except that the set of deleted edges DE is not used. Instead, the graph G is modified before making recursive calls and is restored before returning.

```
TREE:   Procedure (G, NODES, TNODES, ATCH)
        Recursive;
Step 1. If TNODES is empty then add any node I
        to it; ATCH(I) = empty;
Step 2. /*Does the family span G?*/ If TNODES
        = NODES then (call JOB (NODES,
        ATCH); Return;)
Step 3. /*Construct a new attachment set.*/
        Choose node I in NODES-TNODES
        which is adjacent to at least one node in
        TNODES; if there are none then return;
        TEMP = G(I) ∩ TNODES;
        /*TEMP is the attachment set at I*/
Step 4. /*Grow the family of trees which does not
        have edges from I to a node in TEMP.*/
        G
        G(I) = G(I) − TEMP;
        Call TREE (G, NODES, TNODES,
        ATCH);
Step 5. /*Grow the family of trees which does have
        edges from I to a node in TEMP.*/
        G(I) = G(I) + TEMP;
        TNODES = TNODES ∪ {I};
        ATCH(I) = TEMP;
        Call TREE (G, NODES, TNODES,
        ATCH);
        Return;
        End TREE;
```

REMARKS

A recursive procedure sometimes runs more slowly than a corresponding iterative procedure. This phenomenon occurs in recursive procedures which recompute the same quantity in different branches of the recursion. The wasteful recomputation can be eliminated by an exchange of storage for time. By allocating a table for intermediary results, with flags to indicate which entries have been computed, the procedure can retrieve previously computed values. Using this technique, the recursive procedure, except for the cost of procedure calls, is just as fast as a corresponding iterative procedure. Moreover, the added storage is usually just that storage need to implement the corresponding iterative procedure.

Some other problems which are ideal for a recursive approach are: adaptive integration,[3] the towers of Hanoi game (also called the end of the world game), and determining if one multivariable polynomial divides another.

REFERENCES

1 S M JOHNSON
Generation of Permutations by Adjacent Interchanges
Math Comp 17 283-285 1963

2 M D McILROY
to be published

3 W M McKEEMAN
Adaptive Numerical Integration by Simpson Rule
Algorithm 145 CACM 5 604 1962

Statistical validation of mathematical computer routines

by CARL HAMMER *Ph.D.*
UNIVAC Division of Sperry Rand Corporation
Washington, D. C.

INTRODUCTION

Increasing attention is being given to a host of problems associated with the use of the so-called mathematical subroutines in computational processes executed on digital computers. An assessment of the routines available today with most machines indicates a definite need for better validation procedures and for more carefully chosen terms in pertinent documentation. As a matter of fact, what is really required is simply greater adherence to "legal" documentation: to tell the truth, the whole truth, and nothing but the trust.

Measure of precision

A suitable measure of precision for mathematical subroutines is furnished by

$$M(\lambda) = -1/2 \log_\lambda \{E(\lambda^{-\beta(x)} (f(x)-f^*(x)))^2\}$$

where

$f(x)$ = True functional value for all $x \in X$
$f^*(x)$ = Rounded or approximate value produced by the computer subroutine,
λ = Base of computational system,
$\beta(x)$ = Suitable normalization exponent.

Under the assumption of a uniform distribution for rounding errors in the $(\beta+1)$-st digital position of the λ -ary mantissa, we have

$$M(\lambda) = -1/2 \log_\lambda \left\{\lambda^{-2\beta(x)} \lim_{N\to\infty} \left(\sum_{n=1}^{N} n^2/(2N)^3\right)\right\}$$

$$= -1/2 \log_\lambda \left\{\lambda^{-2\beta(x)}/12\right\}$$

$$= \beta(x) + \log_\lambda \sqrt{12}$$

We can identify at once the well-known term $\sqrt{12}$ as the variance of the rectangular distribution. $\beta(x)$ is the position of the last (and also, least significant) "correct" digit of the λ-ary approximation $f^*(x)$ to $f(x)$.

For many computer-used, mathematical subroutines $\beta(x)$ may be assumed constant over fixed ranges $x \in X_j$. For example, if $f(x) = \cos(x)$, then $\beta(x)$ is certainly constant for $0<x<\Pi/4$ which also happens to be the range for which this functional subroutine is often developed.

Monte Carlo evaluation of precision measure

The assumption $\beta(x)$ = Constant does not always hold strictly for the admissible range of the arguments $x \in X$ in the computation of $f^*(x)$. Nevertheless, a suitable approach to estimates of the routine's precision is furnished by the following Monte Carlo approach:

$$M(\lambda) = -1/2 \log_\lambda \left\{\sum_{i=1}^{I} (G^*(f(x_i), f(2x_i), f(3x_i),...))^2/I\right\}$$

where

$$G^*(f(x_i), f(2x_i), f(3x_i), \dots) \approx 0$$

is chosen such that

$$G(f(x_i), f(2x_i), f(3x_i), \dots) \equiv 0$$

and the $x_i \in X$ are randomly and uniformly selected from the admissible space X for all x.

To cite just one example, consider the fact that many practitioners of the computing art used to validate sine and cosine routines by making use of such relations as $G(x)=\sin^2(x) + \cos^2(x)-1=0$, or of $G(x)=\sin(2x) - 2 \sin(x) \cos(x)=0$. However, as we shall see presently, this test does not necessarily furnish anything but a "local" measure of computational accuracy, instead of the desired "global" measure of functional precision. A more pedestrian approach which is used largely during the debugging stage of such routines is the tabular comparison method; but this method is both tedious, time-consuming, and it does not inspire great confidence into the final data quality.

Objectives of validation

The comparison of tabulated and computer values provides the sub-routine designer with some degree of confidence in his choice of algorithms and their correct formulation. However, beyond this level only statistical validation can be used to establish "quality

assurance levels" for the myriads of computations to which this algorithm may be subjected during its lifetime. Specifically, the following objectives are met easily and elegantly through statistical validation:

Range validation of algorithm

Monte Carlo testing can be used to determine (i) whether the algorithm works over its entire design range $x \in X$, and (ii) whether it provides uniform precision overvarious sub-ranges $X_j \in X$.

Characteristic structure of algorithm

Even if an algorithm satisfies the range checks of the above, it could well exhibit characteristic weaknesses at end-points of the subranges $X_j \in X$ or for a set of values $X_i \in X$, not immediately ascertainable from an inductive analysts of the algorithm.

Estimation of precision

The most useful result obtainable from a statistical analysis is the deductive measure $M(\lambda)$ which characterizes the algorithm's quality, even if we don't possess any detailed knowledge about the structure of the algorithm. This aspect is of great importance to computer users who rarely have time or the inclination to study the theoretical approach(es) chosen by various analysts in the formulation of pertinent algorithms.

Selection of test relationships

As we mentioned earlier, not all implicit relationships $G(f(x), f(2x), f(3x), \ldots)=0$ lend themselves to development of useful relationships $G^*(f(x), f(2x), f(3x), \ldots)\approx 0$. Generally speaking, the more complex relationships exhibit a greater propensity toward furnishing good tests. For example, most sine-cosine routines in today's computers reduce the initial argument modulo $\Pi/4$, or better; therefore the usual single or double angle formulas tend to give an erroneous impression about the precision of the chosen algorithm.

Thus we are led to recognize that these test relations are either "friendly" towards the algorithm (and should be avoided), or they are "hostile." Only the latter class of relations satisfies to some extent the conditions implied above. Again, in terms of trigonometric routines, $G(x)=\sin(3x) - 3\sin(x) + 4\sin^3(x)$ would be a good whetstone and more complicated expressions should definitely be tried, at least, initially.

Examples and their analysis

Our first example is taken from the library of trigonometric subroutines available with the FORTRAN compiler on the UNIVAC 1107/1108 computers. Using a random number generator, we selected one thousand random arguments for substitution into $G^*(x)$ which corresponds to $G(x) = \sin(3x) - 3\sin(x) + 4\sin^3(x)$. The results of these substitutions, for various ranges of $X_i \in X$ are shown in Table I.

Table I
Precision estimates for the sine-routine on the UNIVAC 1107/1108

Range of random arguments	$\sqrt{\sum_{i=1}^{I} (G^*(x_i))^2/I}$	Estimated number of significant digits	
		Binary	Decimal
$\lvert x_i \rvert < \Pi/10$	$0.3520 * 10^{-8}$	28.1	8.45
Π	$0.3836 * 10^{-7}$	24.7	7.42
$10\,\Pi$	$0.1997 * 10^{-6}$	22.3	6.70
$10^2\,\Pi$	$0.1670 * 10^{-5}$	19.2	5.78
$10^3\,\Pi$	$0.1969 * 10^{-4}$	15.7	4.71
$10^4\,\Pi$	$0.2022 * 10^{-3}$	12.3	3.69
$10^5\,\Pi$	$0.1708 * 10^{-2}$	9.2	2.77

The table indicates very clearly that the chosen algorithm loses precision as the arguments become larger. In retrospect, this weakness is easily traced to the method chosen for reducing arguments modulo $\Pi/4$ to allow entry into a tightly coded routine. Of course, this modular reduction is only carried out to limited accuracy (exceeding the range of the usable 27-bit mantissa by 8 bits) and its effects become quite noticeable for larger arguments. On the other hand, the table also indicated that the basic algorithm meets its design goal well for small values of x.

A second example is given in Figure I. Here we compare the results, in terms of "apparent" significant digits, both decimal and binary, obtained for several multiple-argument relations. Only "hostile" relations such as the triple-angle formulas provide information about the precision of the algorithms; "friendly" relations involving only single and double angles must be ruled out here from further considerations.

Fortunately, greater precision in the sine/cosine subroutines can be attained by using double-precision coding. Furthermore, the angular reduction modulo $\Pi/4$ could certainly be accomplished in double or even triple precision, if desirable, even in the single precision routine.

Finally, we might list several useful relations which are typical of tests on elementary or advanced functions. For additional details, we refer our readers to Reference (1).

Examples of elementary functional test relations

Exponential Function: $G(x) = e^{3x} - (e^x)^3$
Logarithmic Function: $G(x) = \log(x^2) - 2\log(x)$
Hyperbolic Functions: $G(x) = \sinh(2x) - 2\sinh(x)\cosh(x)$

Examples of advanced functional test relations

Gamma Function: $G(x) = \Gamma(1 + x) - x\,\Gamma(x)$
Bessel Functions: $G(x) = J_{v-1}(x) + J_{v+1}(x) - 2v\,J_v(x)/x$

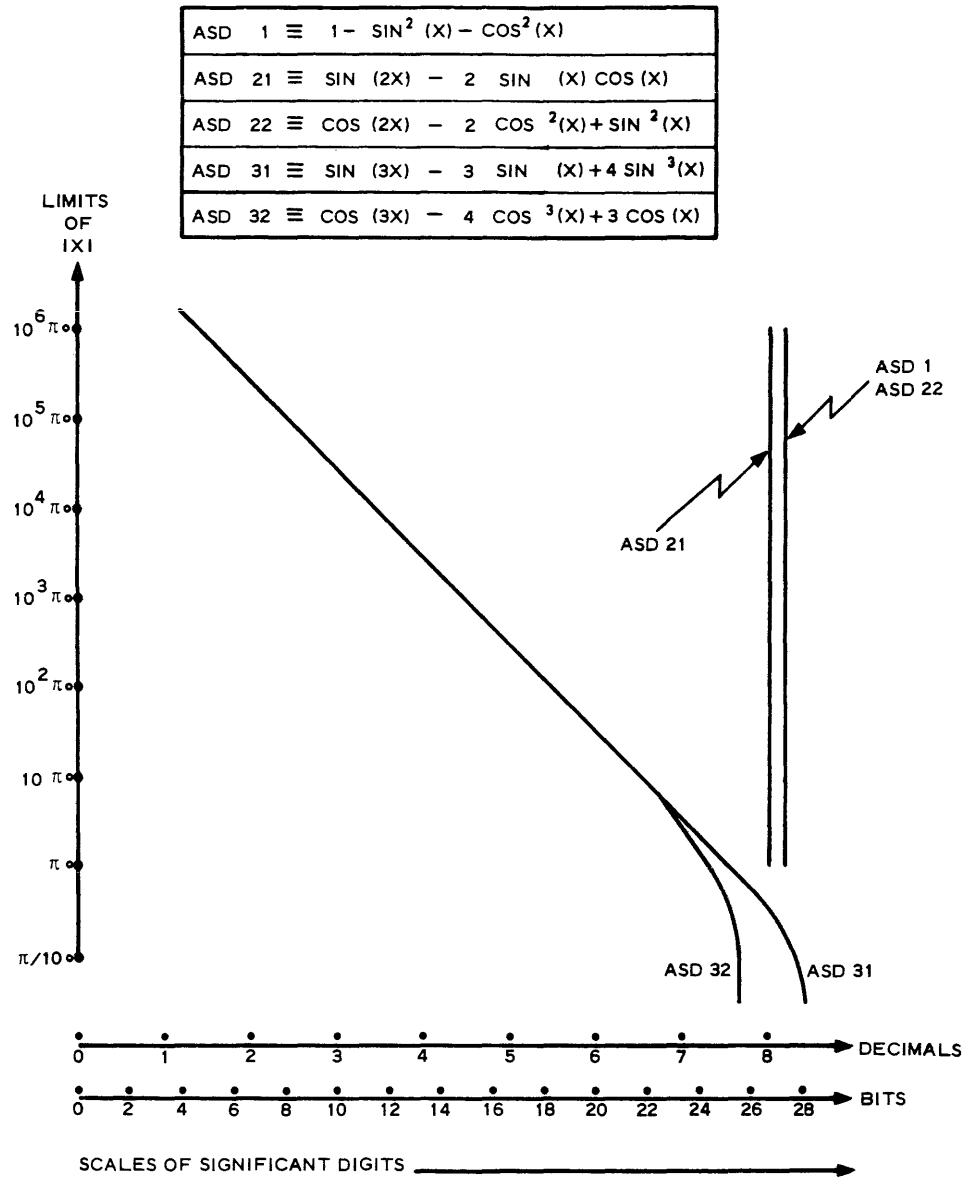


Figure 1 – Validation of computer routines

Credibility of computational results

As computer operating systems and compilers assume greater degrees of complexity, large-scale computational efforts become more and more a matter of faith. Users often rely on (possibly meager) documentation, if it is at all available; or on intuition and other sources of questionable value. We have tried to show here that such need not be the case. By exercise of some ingenuity, computational results of the most involved operations can be subjected to analysis and assessment. In fact, this approach is highly recommended where computer outputs may become the basis for far-reaching decisions. In this vein, credibility of computation becomes then itself a subject of intensive study and investigation. As shown by our examples, there exist deductive methodologies which may be invoked independently from inductive analysis.

REFERENCES

1 H BATEMAN
Higher transcendental functions
McGraw Hill Book Company Vol I-III 1953
2 A S HOUSEHOLDER
Principles of numerical analysis
McGraw Hill Book Company 1953
3 M G KENDALL
The advanced theory of statistics
Hafner Publishing Company Vol I&II 1952
4 A RALSON S WILF
Mathematical methods for digital computers
John Wiley 1962

Macromodular computer systems*

by WESLEY A. CLARK
Chairman's Introduction
Washington University
St. Louis, Missouri

INTRODUCTION

The amount of logically irrelevent engineering detail inherent in the design and construction of a computer system is great. As a result, the task of creating a system based on the use of present techniques is so difficult and time-consuming that the number of different systems that can be put into use for evaluation and study by any one group of workers is small. This is unfortunate as we are thereby denied the opportunity to develop that insight into logical organization which can grow out of a working familiarity with many diverse forms. What is needed is a set of relatively simple, easily inter-connected modules from which working systems can be readily assembled for evaluation and study. With such a set, both the designer and user would be able to try out potentially powerful and novel structures on a very large scale, adjusting and improving the systems as needed. Once a design has been realized and its value established, it could then be reworked into tighter engineering form for maximum efficiency and for production by automatic wiring and fabrication techniques, and the experimental units made available for further studies or returned to "inventory" in the manner proposed by Estrin.[1]

The approach presented in the "following", papers describes modules which are primarily vehicles for experimental use and as such meet a set of requirements heretofore unnecessary in digital modules. Logical flexibility and ease of use are considered of primary importance while factors such as operating speed, economy, etc., are considered of secondary importance. The requirements can be summarized as follows:

(1) The modules must be functionally l a r g e enough to reduce logical detail by a significant amount and must be relatively easy to understand and assemble. The number of different types should be as small as possible so as to limit inventory, but at the same time, the set must be logically complete so that whole systems can be assembled. There must be not only central processor modules such as register and memory units, but also modules for power, signal conditioning, input-output buffering and control, together with a reasonable selection of input-output devices themselves.

(2) The mode of combining units into larger structures must be very simple (a problem first considered by Babbage, who examined this matter "with unceasing anxiety" one hundred and twenty years ago).[2] The modules should be designed for easy mechanical assembly. Communication from one mechanical assemblage to another should be accomplished by means of easily connected cables.

(3) All units should be designed so that the assembling of these units into a working system presents no logically irrelevant problems such as those relating to circuit loading, waveform deterioration, signal propagation delay, power supply interactions, and so forth, *regardless of the size or complexity of the system.* The modules should be powered and perhaps controlled individually, and all possible signal paths must be provided with signal-standardizing amplifiers capable of driving all possible loads.

We call units which meet these requirements *macromodules* to distinguish them from the more

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

conventional computer system modules. In this report we present a set of macromodules which, although not "complete" in the above sense, meets all other requirements and is sufficient for the synthesis of all central processor functions of which we are presently aware.

The following papers present the principal direction, achievements, and goals of the macromodular computer development program under way at Washington University. They describe an approach in which flexibility and simplicity are brought to the forefront of factors relevant in computer design. Two aspects of this approach are distinguishable.

First, it adds an experimental element to the theoretical and simulational techniques now available to the system designer. Perhaps this can most effectively be employed in research groups having as their main objectives the creation of advanced computer systems for particular areas of work. Because of the relative ease with which a given configuration of macromodules can be profoundly altered, it is possible for such a group to work actively with several widely different forms in an attempt to find optimal configurations for different problem classes.

Second, it makes possible a smoothness of growth and refinement in an operating computing system. Because of the electronic independence of the macromodules, it is relatively easy to expand a macromodular system and to add new functions without seriously affecting the continuity of on-going work and without jeopardizing any existing investment in programs and operating procedures.

These properties of macromodular systems are of growing importance as we turn increasingly toward the user for new concepts in the search for more effective information processing systems.

REFERENCES

1 G. ESTRIN
Organization of computer systems the fixed plus variable structure computer
Proc. WJCC 33-40 1960

2 L. F. MENABREA
Statement of the circumstances attending the invention and construction of Mr. Babbage's calculating engines
Philosophical Magazine 235 September 1848

A functional description of macromodules*

by SEVERO M. ORNSTEIN, MISHELL J. STUCKI and WESLEY A. CLARK
Washington University
St. Louis, Missouri

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

INTRODUCTION

This paper describes a set of macromodular building blocks such as registers, adders, memories, control devices, etc., from which it is possible for the electronically-naive to construct arbitrarily large and complex computers that work. Machines are assembled by plugging the modules into cells of a special frame which provides for communication between adjacent cells. Explicit data pathways and control structures are then made by plugging in standardized cables. All pieces of a system are therefore recoverable and systems can be reconfigured easily. Data modules process twelve-bit word-segments; greater word lengths are obtained by interconnecting modules. Memory modules hold 4096 twelve-bit segments and can also be interconnected to form larger arrays. Particular attention is given to the problem of designing control structures. The control signals for a given process are routed along the cables of a control network whose topology is isomorphic to the flow diagram representing the process. The step from conception to realization can therefore be made directly.

The task of defining a set of macromodules or building blocks is not unlike that of defining an instruction repertoire for a computer. The fundamental requirement is that the set be sufficiently general to permit construction of any central processor.* The particular set described here embodies this generality and satisfies the requirements set forth in another paper.[1] While it illustrates the approach we have taken, the set is by no means unique and, like a particular instruction code, will be more convenient or efficient for some tasks than it will for others.

General characteristics

The macromodules to be described are relatively small, dimensionally modular boxes which plug into a cellular frame structure, some modules occupying more than one cell. Each module contains all of the electronic circuits and memory elements required in the performance of its particular function. Connectors on the frame provide for communication between modules in neighboring cells, and assemblages of units are thus made by plugging them into appropriately adjacent positions. Faceplates attached to the modules' front surfaces provide the electrical connectors for signal access, and standardized cables are provided for inter-assemblage communication. All connectors are backed by signal-standardizing amplifiers capable of driving any adjacent module or attachable cable. Since all cabling takes place between faceplates which are separable from the modules, it is possible to remove modules from a frame for temporary use elsewhere without disturbing the cabling.

Data processing modules are organized in parallel binary form with a word-length modulus of 12 bits, and are designed functionally for asynchronous operation. Memory modules hold 4096 12 bit words.†

The design of a system based on these modules requires, we believe, only the exercise of logic. The operability of the resulting system is not critically affected by the physical distribution or arrangement of

*No input-output macromodules are discussed here although modules for various devices (scopes, tapes, printers, readers, etc.) are obviously required to complete the set.

†The numbers 12, 4096, and other such parameters have been made specific, for purposes of this paper, only to simplify description.

parts, the distances between units, the number or diversity of modules, or the routing of the interconnecting pathways. Macromodular systems are, as a result, capable of continuous growth and functional enrichment.

System organization

Macromodular systems may be viewed in terms of two logically distinct, interacting networks as shown in Figure 1. The *processing network* (the heavy-lined structure) consists of data processing elements interconnected by data pathways, and provides for the storage, propagation, and transformation of data within a system. The *sequencing network* (the light-lined structure) consists of control nodes distributed throughout the system, interconnected by control pathways. The structure of the processing network defines the basic data processing operations of the system while the structure of the sequencing network defines the order in which subsets of these basic operations can be carried out.

Interaction between these networks takes place at control terminals on the data processing elements. These terminals have two functions:)1) they allow the sequencing network to initiate operations, and (2) they return completion signals when the operations are finished. Each basic data processing operation has an associated set of these terminals (Figure 2), the number of terminals in the set being determined by the nature of the operation. Operations that manipulate data, *data operations*, have two, an initiation terminal and a completion terminal. Operations that check data for specific values, *decision operations*, have more than two, one of which initiates the operation while the others (completion terminals) indicate the value of the data found. Also shown in Figure 2 is a *time continuous transformation* element. This element, unlike those already described, performs its operation continuously. The data presented at its output changes directly in response to changes of input data rather than in response to control signals, and as a result, the element has no control terminals at all. An operation is initiated when a control signal arrives at an initiation terminal. The operation is executed and finally a control signal issues from the completion terminal and travels to the next control node in the sequencing network.

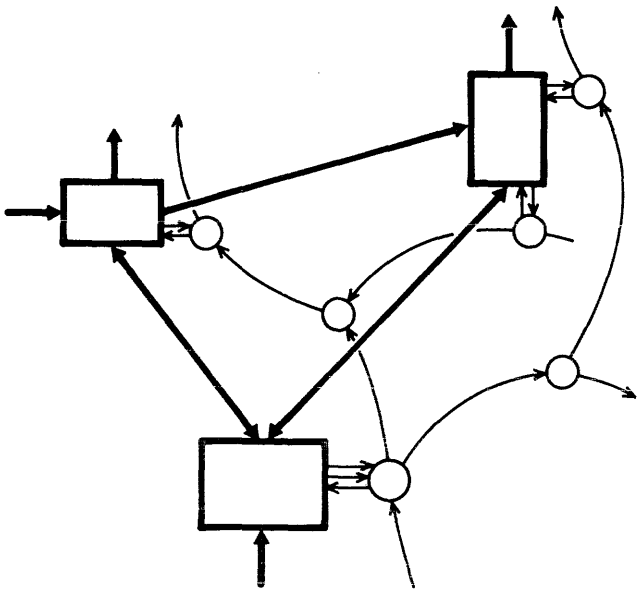


Figure 1 – Processing and sequencing networks

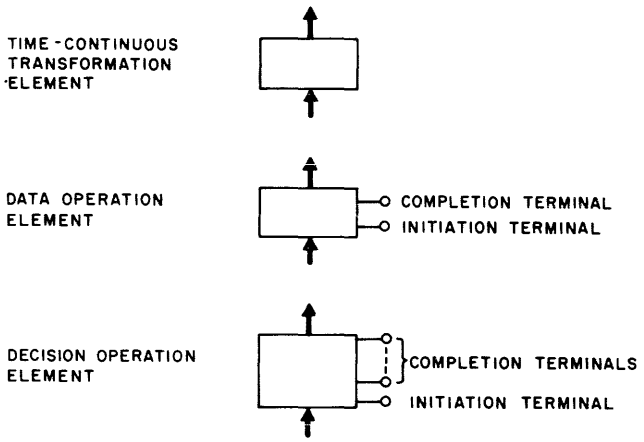


Figure 2 – Data processing operations

The order or sequence in which operations are performed is determined entirely by the structure of the sequencing network. This network is composed of signal nodes, calling elements, and interconnecting pathways. A *signal node* is an element which provides for the merging or branching of control signals. There are several types, two of which are shown in

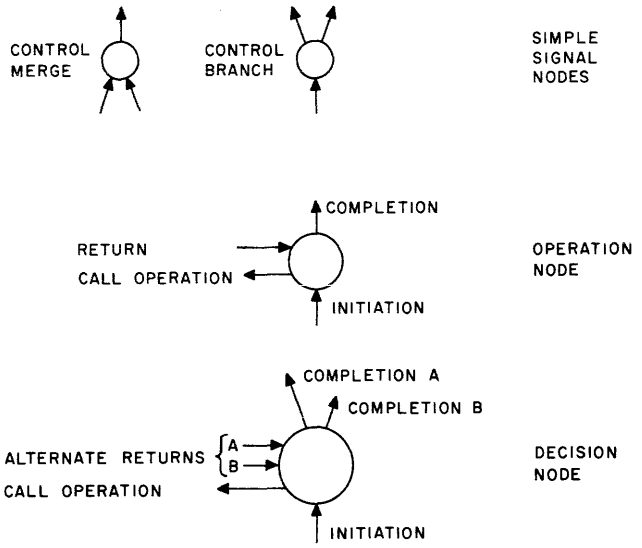


Figure 3 – Control nodes

Figure 3. A *calling element* is one which, when activated by a control signal at its initiation terminal, causes an operation to take place and, when signaled of completion of the operation, produces its own completion signal in turn. An *operation node* is a calling element for data operations, and a *decision node* is a calling element for decision operations.

Control within a macromodular system is asynchronous, that is, each event in a sequence of events can be initiated by the completion signal from the preceding event. The simplest way of accomplishing this is to connect a cable from the completion terminal associated with each operation to the initiation terminal associated with the next operation. This scheme, though simple and effective, has the limitation that once the control terminals for an operation have been connected for one sequence, it is no longer possible to incorporate the operation into any other sequence. In such cases, rather than connect to the terminals associated with the operation, we connect instead to the terminals of a calling element associated with the operation, as shown in Figure 4. Since any number of calling elements may call the same operation, an operation may thus occur in as many distinct sequences as necessary. Figure 4 illustrates this for two different sequences, namely, the sequence z, y, x and the sequence z, x. Since both sequences include the operations z and x, they initiate the operations through calling elements. Calling elements are not needed for operation y, however, as it appears in only one sequence and can therefore be incorporated by connections directly to its control terminals.

The control elements and interconnections defining a given sequence are said to be the *control path* for that sequence. The sequencing network is therefore the control path for the entire system.

Data validation

Whenever data values are used in either a data operation or a decision operation, it is necessary to be assured that 1) the results of all prior operations which could have perturbed the data are complete and 2) the new values of the data have propagated to the point of use regardless of the length of the pathway.

When information is used in the immediate locality of its source, i.e., within the same module, allowances for stabilization and signal progagation times are made within the module itself. When the source is remote from the point of use, a procedure known as *data validation* is followed to guarantee that the above two re-

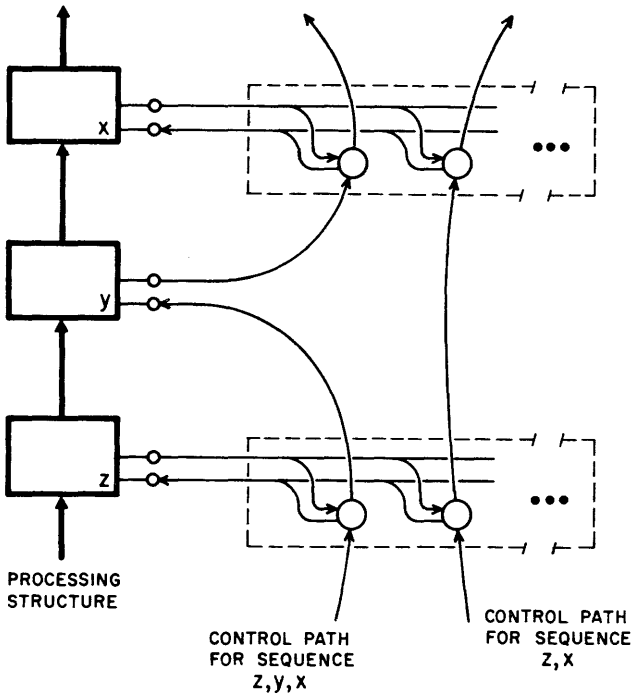


Figure 4 – Example of sequence control

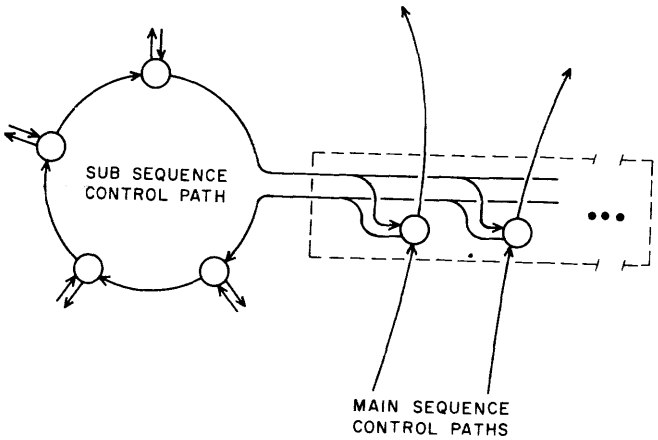


Figure 5 – Subsequence control

quirements are met. This process is discussed elsewhere.[2] For the present purposes, we shall assume that any attempt to use data following a perturbation of the data source will work properly, i.e. that the correct new value of the data will be used. Thus, there is no need for the designer to concern himself with details of propagation times so long as proper sequence is established.

The macromodules

We now proceed to give a functional description of the individual macromodules and illustrate their roles in various systems. Processing network elements are introduced first, and this is followed by a discussion of the various sequencing network elements. Power connections and the supporting frame structure are omitted from the figures to avoid obscuring the logical point being illustrated. A basic module type is sometimes fitted with more than one type of faceplate suiting it to different contexts. In such cases the circuits within the module sense the faceplate type and operation is suitably adjusted.

Cables

Data paths are constructed with *data cables*, control paths with *control cables*. These cables are made in a limited number of lengths, but cables of any length can be formed by using signal-standardizing extender units. A control cable contains a single channel for transmission of a control signal. A data cable contains 12 channels for the transmission of data and two for the transmission of signals associated with data validation. In the illustrations, data cables are drawn with heavy lines and control cables are drawn with thin lines.

Input switch sets

Any faceplate data input connector will accept either a data cable or an *input switch set*. The switch set is used to provide "constants" for presetting registers, masking, and so forth.

Registers

The basic register module Figure 6 contains a 12-bit register together with logic for the operations *clear, complement,* and *index* (count). Mounted on the faceplate of the register are control terminals for these operations together with a data output connector. Outputs are also carried up to the overlying cell. Data input to the register comes from units plugged into the underlying or overlying frame cells.

Registers of any length can be formed by plugging these modules into laterally adjacent cells in the frame (Figure 7). Only the faceplate for the rightmost module bears control terminals, and these terminals provide control for the whole register. This feature is common to all data operation and decision operation modules, thereby making control of an operation independent of register size. Special circuits within each module are coupled in such a way as to guarantee proper operation regardless of register length.[2]

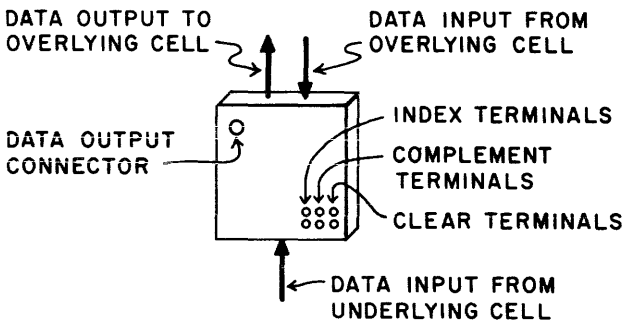


Figure 6 – Register module

Transfer operations

Data transfers from one register module to another require the use of a *data gate* module. This unit plugs into the frame cell underlying the receiving register and is connected by means of a data cable to the output of the data source register module (Figure 8). Twelve bits are transferred in parallel, and the transfer initiation and completion terminals appear on the data gate module. Transfers do not alter the information at the source.

If a register module is to receive input from n sources, n data gates are required (Figure 9). Stacking the units in this way allows each data gate to communicate with the receiving register module; any number of transfer paths into a register module can be provided. Two registers cannot exchange information without the aid of a third register, however, since simultaneity of events in different parts of an asynchronous system cannot be assumed.

Data gates plugged into laterally adjacent cells form *tiers* which provide for transfers into longer registers. Figure 10 illustrates a 24-bit transfer. Interconnect-

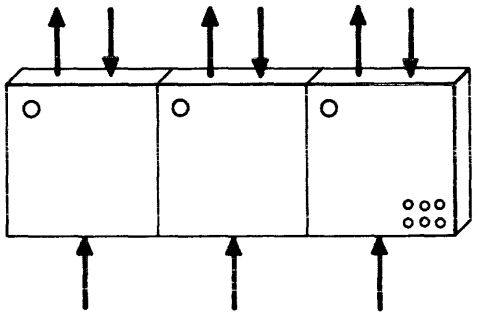


Figure 7 – Register extension

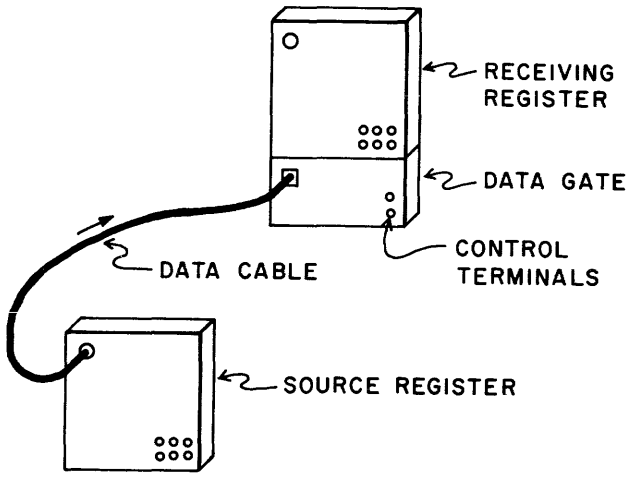


Figure 8 – Data transfer

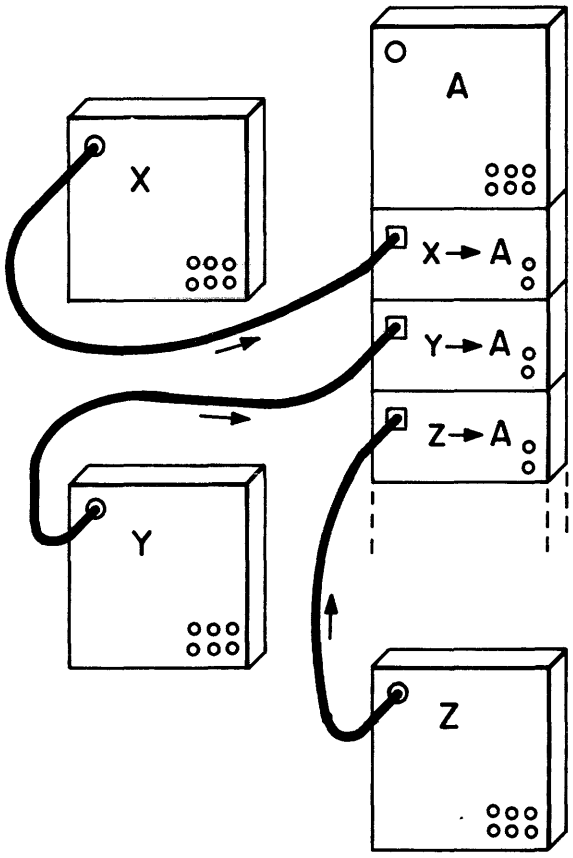


Figure 9 – Multiple transfers

ing data cables for the register segments may be of different length, as compensation for signal propagation times is automatically made in each cable.

In order to permit the transfer of information from a single source into more than one destination module, a *data branch* unit is used (Figure 11). Data branch units may be cascaded indefinitely to provide any number of connections to the same source. As it is not necessary to use both outputs, the data branch also doubles as an extender unit for data cables.

Memory

The memory module has a capacity of 4096 12-bit words and contains, in addition to a memory array, all required drivers, addressing logic, sense amplifiers, internal address and data registers, etc. Figure 12 shows a simple arrangement in which one memory module is used.

Reading is controlled by a pair of terminals on the module. In this example the memory module data output terminals are connected so that the word obtained from the memory array can be transferred into register B. The read operation's completion signal or any subsequent signal may be used to initiate this transfer operation.

Writing into the memory may take place from an arbitrary number of sources and for this purpose data gates are stacked in underlying cells just as for register transfers. Writing from each source is controlled by the terminals on the corresponding data gate.

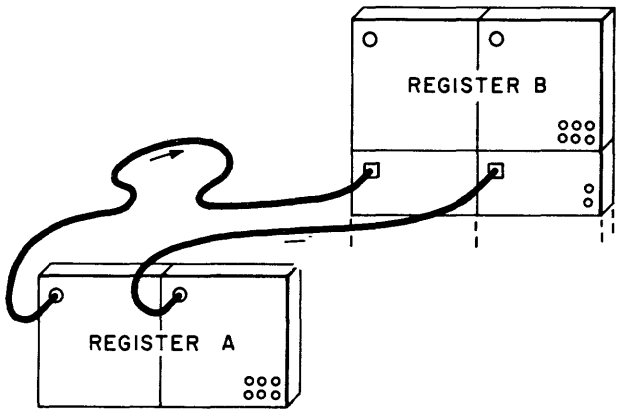


Figure 10 – Double length transfer

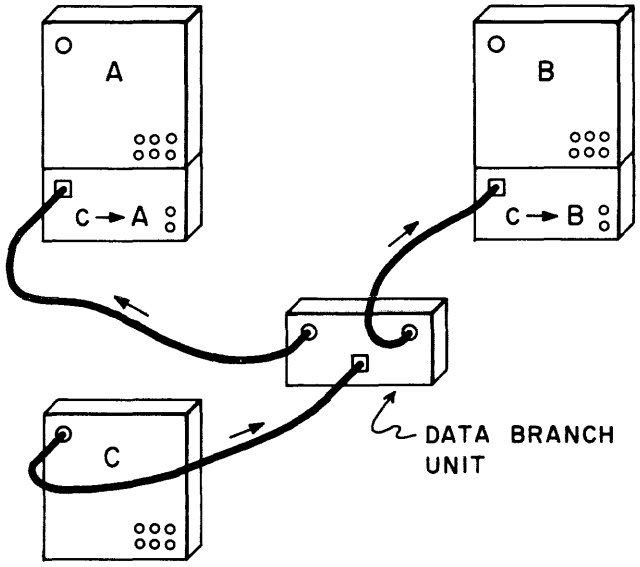


Figure 11 – Data branching

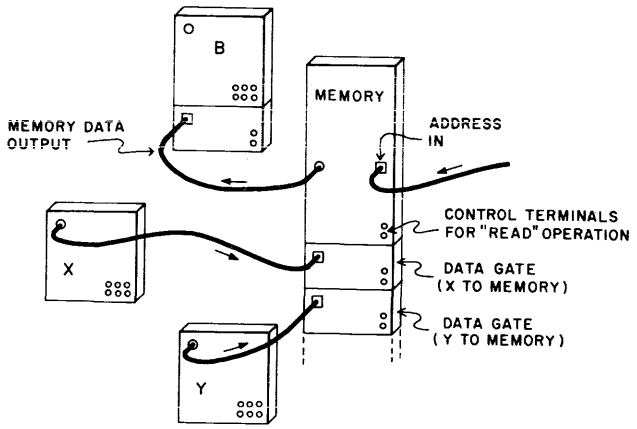


Figure 12—Simple memory

Memory modules can be plugged together laterally to increase word length and vertically to increase the number of words. Figure 13 shows a memory system containing 8192 thirty-six bit words. To permit referencing in memory systems containing more than 4096 words, connectors are provided for additional address inputs for selection of the appropriate tier. All access to the memory, regardless of address, is through terminals on the lowest tier.

Junction unit

A *junction unit* is a continuous transformer which permits one to rearrange bits within a word or to form words from bits selected from several words. The unit has two 12-bit data inputs and a single 12-bit data output. A set of jumpers can connect each of the 12 output terminals to any of the 24 input terminals or to fixed terminals supplying the value "1" or "0".

Suppose, for example, that information is to be transferred from parts of two 12-bit registers, A and B, into a third register, C. Specifically, suppose that the rightmost four bits of A are to be copied into the leftmost four bit positions of C, the leftmost four bits of B into the rightmost four bit positions of C, and the four middle bits of C are to be set to the binary value 1011. The units are arranged as shown in Figure 14, and the junction unit jumpers as shown in Figure 15. The word made up by the junction unit is transferred into C via a data gate.

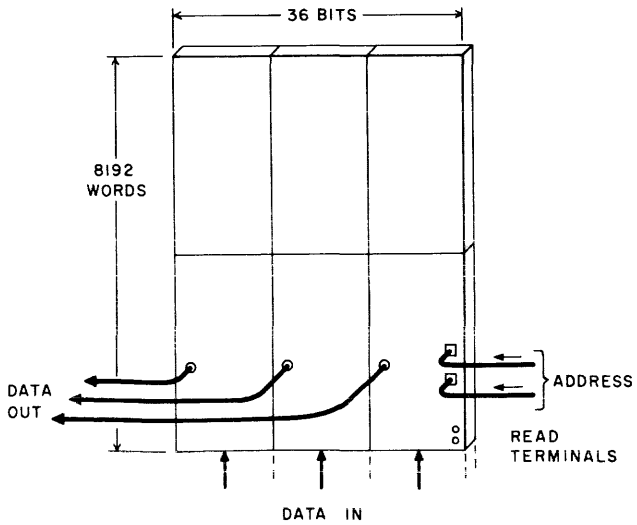


Figure 13—Memory extension

Data input from overlying units

Three types of units, namely, Shifters, Adders, and Function Units, overlie a register (or one another) and transmit their outputs only downward to the register via the implicit frame data pathway. These units have the following properties in common:

1 Each unit passes the data from the register on up to overlying cells and similarly, provides upward continuation of the down access route into the register.
2 Each unit operates in response to control signals presented to terminals on its faceplate.
3 Each unit uses the information originally held in the register below in determining the results to be returned to the register.
4 Each unit extends laterally as the register length extends, forming tiers which overlie the register.

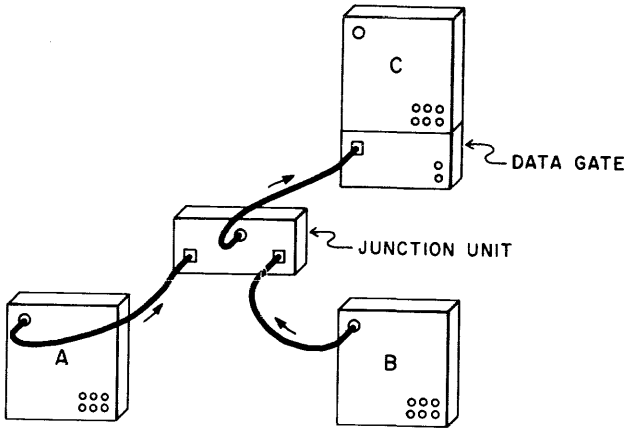


Figure 14—Junction unit

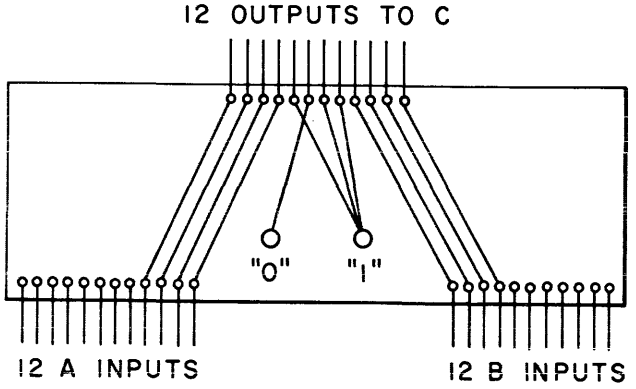


Figure 15—Junction unit jumpering

As many tiers as desired may be stacked on top of one another above the register. Each tier must extend across the full length of the register and must consist of only one type of unit. An example is shown in Figure 16.

Shift unit

Two types of shift modules are defined, one for shifting right and one for shifting left. Figure 17 shows a 36-bit register equipped to shift in either direction.

Each of five pairs of control terminals causes the register to shift one position, but each pair treats the incoming bit at the trailing end of the register differently. The options are as follows:

1 The bit is not changed.
2 The bit is replaced with the bit previously at the other end of the register, i. e. a rotation occurs.
3 The bit is set to the value "0".
4 The bit is set to the value "1".
5 The bit is taken from the data input port at the trailing end of the register. For leftward shifting the bit is taken from the most significant bit position of the input; for rightward shifting, from the least significant position. (The remaining 11 bits of information at the data input are not used.)

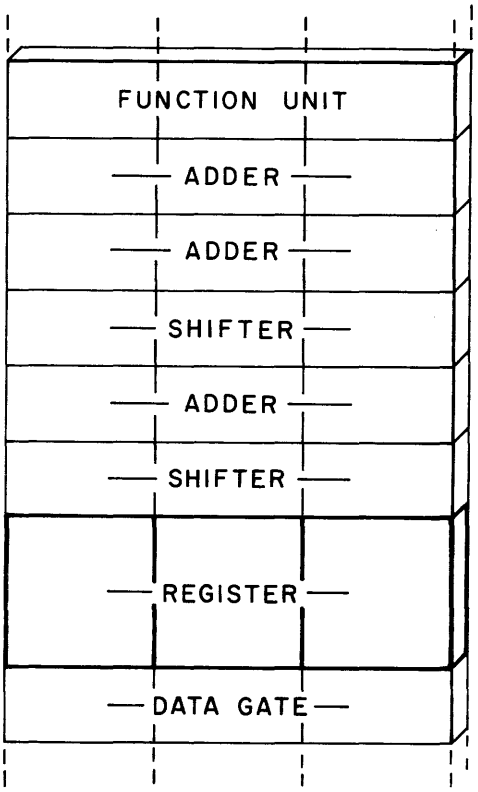


Figure 16 – Overlying units

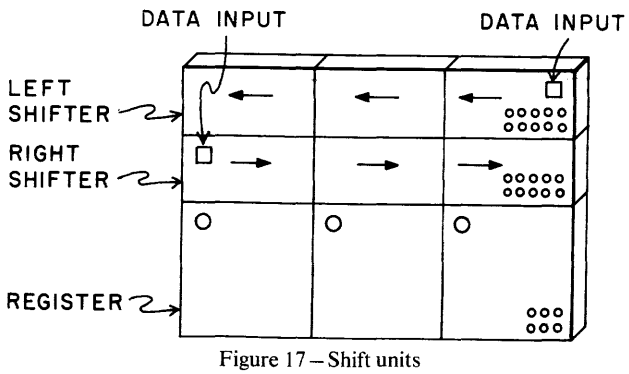


Figure 17 – Shift units

Adder unit

The adder unit takes one input from the underlying register and the other input from a data input port on the adder's faceplate. The sum is copied into the underlying register. Addition is controlled by terminals on the rightmost module in an adder tier. Figure 18 shows a 24-bit register, A, equipped to add from registers X or Y, (i.e. A + X→A or A + Y→A).

The adder unit contains three decision nodes which provide for the detection of overflow, negativity and the numerical zero sum. Their use is discussed below in the section dealing with control decisions.

Function unit

A function unit may perform any of three logical operations on a pair of data inputs. Like the adder, one of the data inputs comes from the underlying register and the other from a data input port on the front of the unit. The result of the operation is returned to the underlying register. The operations, controlled by three pairs of terminals on the rightmost unit, are the logical "OR" (v), the logical "AND" (·), and the "EXCLUSIVE OR" (v) Figure 19 shows a single 12-bit register equipped with a function unit as well as an adder.

An example of the use of the function unit might be the clearing of selected bits of the register. An input switch set may be inserted in the data input port of the function unit. If the switches are set to the value 1777_8, then whenever the "AND" terminals

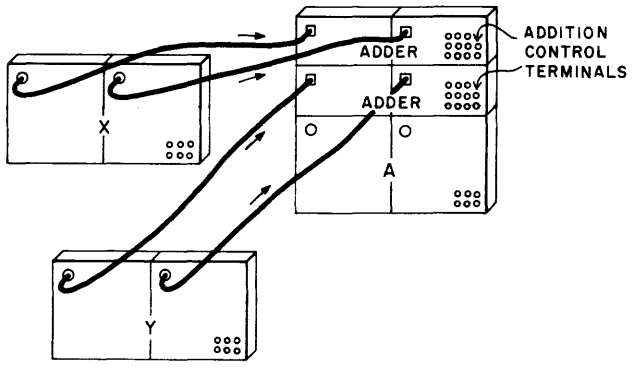


Figure 18 – Adder units

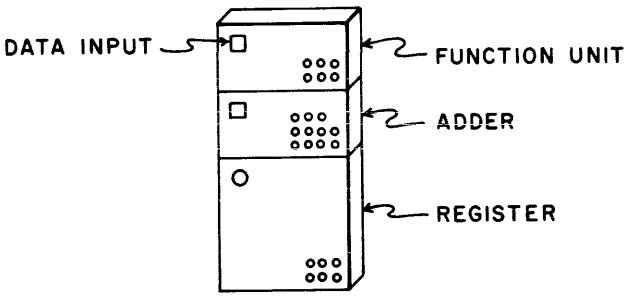


Figure 19 – Function unit

receive a control signal, the leftmost two bits of the register will be cleared.

Sequencing

In order to perform a desired sequence of operations, control signals are routed along control cables from one set of control terminals to the next set in a manner reminiscent of the plugboard programmed machines or the Bell Computer Model VI.[3] Thus, to perform the sequence

A → B
↓
INDEX B
↓
B → C
↓

one would interconnect control terminals as shown in Figure 20.

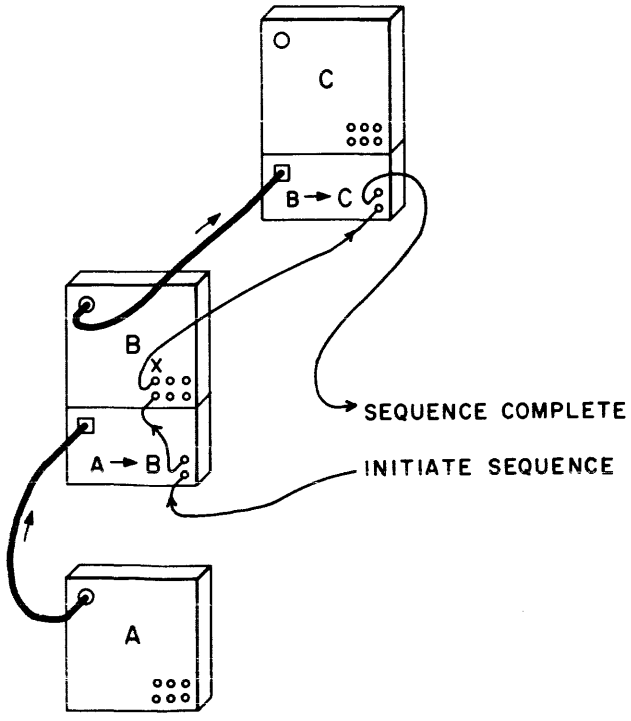


Figure 20 – Sequencing of operations

Concurrent sequences

Use of a control branch module (Figure 21) makes it possible to perform sequences of steps which may be executed concurrently. A control branch module contains several identical control branch elements. A control signal presented at the input terminal of such an element causes control signals to appear on each of two output terminals. These elements may be cascaded to form an arbitrary number of control path branches. They may also, of course, be used simply to extend control cables.

In the execution of two concurrent sequences, there will be found a point at which ensuing steps can be taken only after all steps of both sequences have been completed up to that point. A rendezvous element (Figure 22) which produces a signal (Z) at its output terminal only after signals have arrived at both of its input terminals, (X and Y) is used at the point of conjunction. Like the control branch, several elements are housed in a rendezvous module.*

For example, suppose a problem requires several set-up steps, one of which transfers data from register X to register A, and another of which transfers data from register Y to register B. These steps may either be executed sequentially (Figure 23), or they may be executed concurrently (Figure 24). In the latter case, both transfers are activated and can take place at more or less the same time. As each

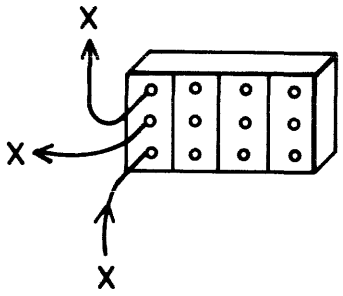


Figure 21 – Control branch module

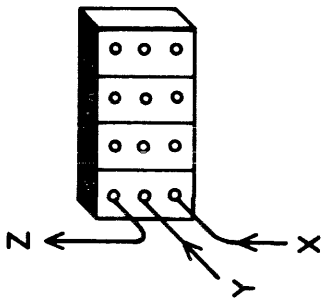


Figure 22 – Rendezvous module

*The rendezvous module is shown with a darkened top to distinguish it from units of a similar appearance.

transfer is completed, a signal is sent to the rendezvous unit. When both signals have arrived, the unit sends out a signal which proceeds on to the next step. A signal indicating the completion of an arbitrary number of concurrent actions can be generated by cascading rendezvous units.

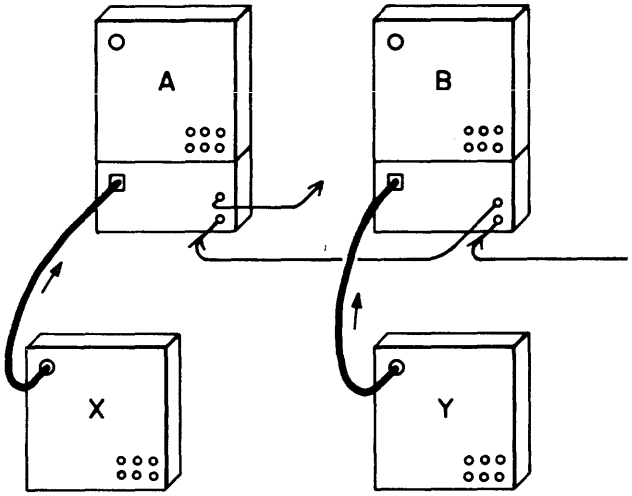


Figure 23 – Sequential transfers

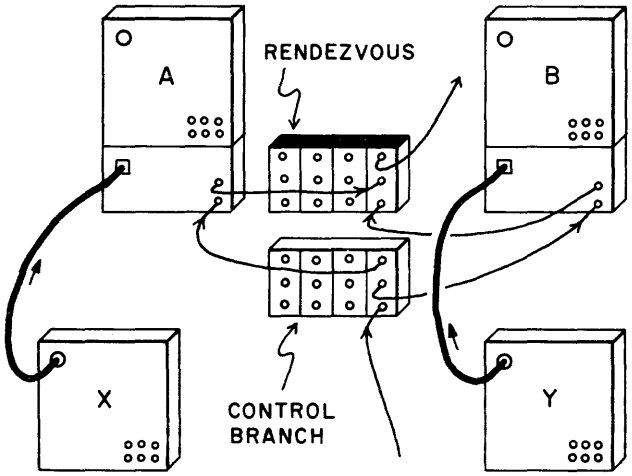


Figure 24 – Parallel transfers

Call unit

For situations in which more than one control path must have access to a single pair of control terminals, calling elements are used. Four calling elements are included in a single call unit.

A call unit is provided with the terminals shown in Figure 25. Terminals 1, 2, 3, and 4 are input terminals for the elements, 7, 8, 9, and 10 are output terminals. Whenever a control signal arrives at the input terminal of one of the calling elements, a control signal is presented at terminal 5 which thus initiates the operation. When the completion signal from the operation is returned to terminal 6, the call unit, in turn, produces a

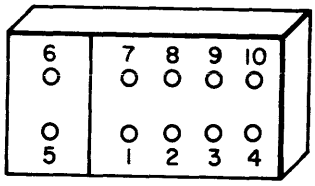


Figure 25 – Call unit

completion signal at the output terminal of the particular element which called for the operation. Call units may be attached to the control terminals for any operation. Figure 26 shows a call unit connected to the complement control terminals of a register module.

Call units may be cascaded, as shown in Figure 27, to increase the number of accesses for a particular operation to any desired number. Figure 28 shows three control paths, two of which contain a step which transfers A to B and two of which index A.

The three sequences performed are:

(1)	(2) •	(3)
COMPLEMENT A	*INDEX A*	*INDEX A*
A→B	A→B	
COMPLEMENT B	B→B	

No call units are required for the complement A, complement B, or B→C operation terminals, as each of these operations occurs in only one of the above sequences.

Sub – sequence calling

Call elements may be used to execute a subsequence common to several main sequences (for example, an operand fetch sub-sequence common to several instructions). After completion of the steps of the sub-sequence, each main sequence must continue with its own set of steps (corresponding perhaps to different instruction steps).

Figure 29 indicates, on the left, the steps of the subsequences S_1,S_2, and S_3. All of the instructions signal

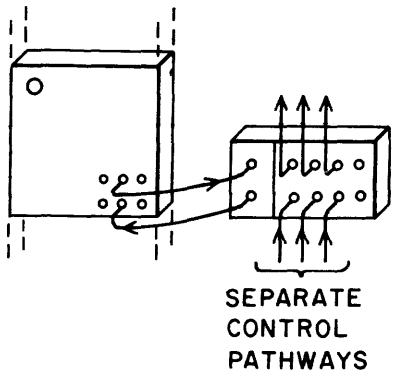


Figure 26 – A use of the call unit

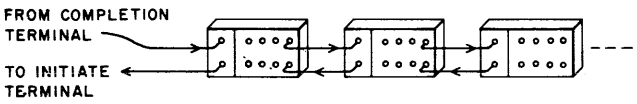


Figure 27 – Extension of call units

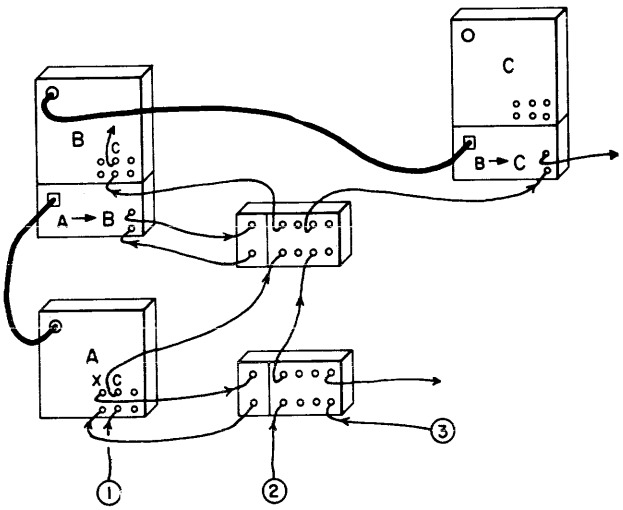


Figure 28 – Example of call unit use

the call unit assemblage on the right at the point in the sequence at which they require an operand. After all steps are completed, a signal is returned to the main calling assemblage, from which each instruction's control signal proceeds to initiate succeeding steps defining that particular instruction. Essentially, then, a call unit remembers which main control path is calling for the performance of a step or a set of steps during the execution of those steps.

Control decisions

In order to permit the choice of alternative steps to be made on the basis of data held by the system, two processing network elements, a detector unit and a decoder unit, are provided.

A detector unit is used to detect a specific value on a data path. It may be plugged into a cell overlying the register which provides data, or alternatively it may be connected via a data cable to a source of data. The binary value to be detected is entered in a set of 12 switches on the unit. A third setting of each switch allows one to indicate indifference to the value of the bit at that position.

A detector unit has three control terminals, one for interrogation and the others to indicate the result. When a control signal interrogates the detector unit, the data is compared with the pattern set in the switches. If the pattern matches the data, a control signal will be presented at the "Yes" terminal. If the pattern does not match the data, a control signal will be presented at the "No" terminal.

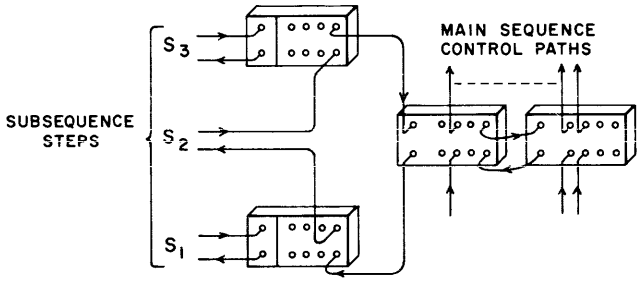


Figure 29 – Sub-sequence calling

Figure 30 shows a detector which tests for the value 101110 in the rightmost 6 bit positions of register A. X's indicate indifference to the leftmost 6 bits. In this configuration the detector unit receives its data input from the underlying register. Likewise, the detector unit passes the data on upward to the overlying cell. Detector units may thus be stacked one above the other in adjacent vertical cells (Figure 31) making it convenient to test for any of a variety of possible patterns of interest. Note that a shifter and an adder unit intervene between the register module and the detector stack. This is permissible in that these units also pass the register outputs upward. It is not permissible, however, to place an adder, shifter or function unit *above* a detector unit inasmuch as these units require downward access to the register for depositing their results, a feature not required by or incorporated in the detector unit.

Figure 32 shows the same stack of detectors, placed in cells not overlying the register. In this case, the source register outputs are delivered to the bottommost detector unit via a data cable.

Detector units extend laterally in the usual fashion for the detection of patterns of more than 12 bits. Figure 33 shows an arrangement which detects a pattern of 36 bits from a variety of sources.

The adder unit, as mentioned earlier, has three sets of detector terminals (Figure 33). These terminals are similar to control terminals on the detector unit and

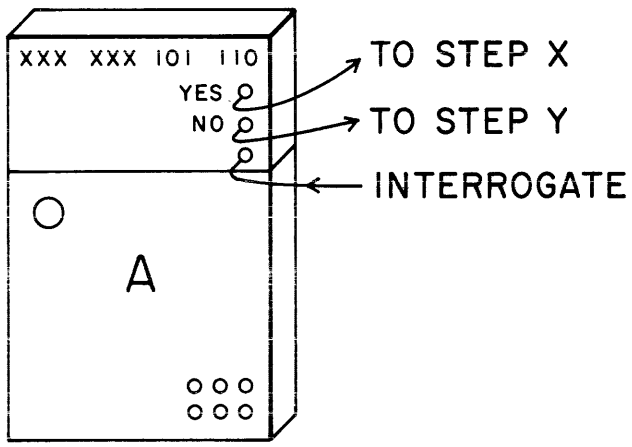


Figure 30 – Detector unit

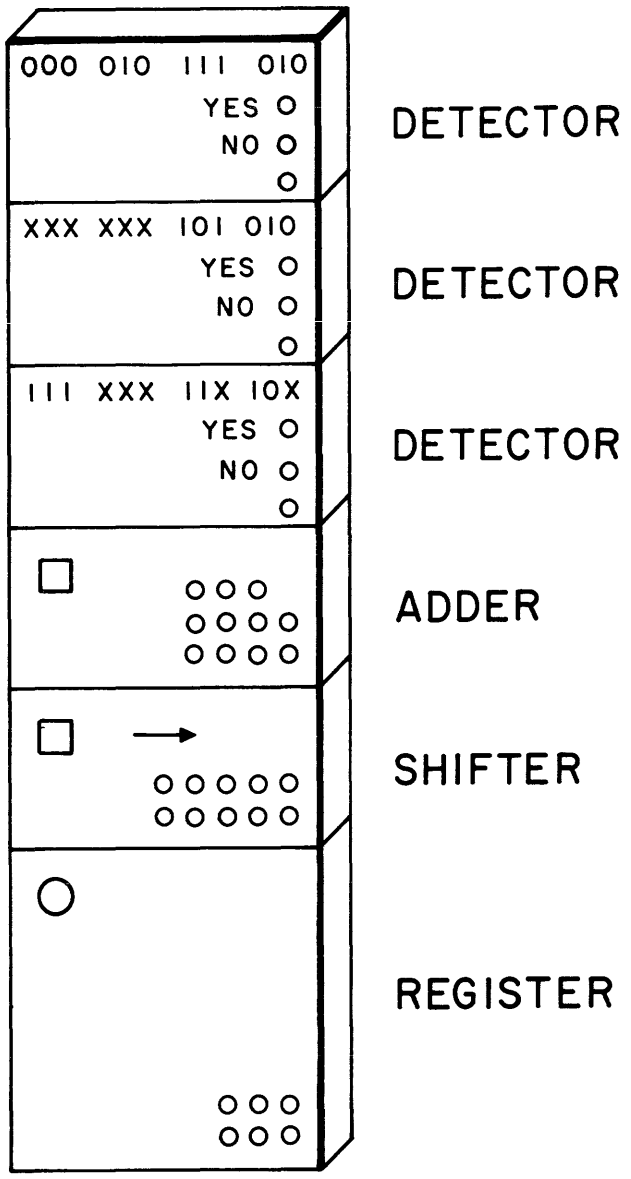


Figure 31 – Stacking of units above the register

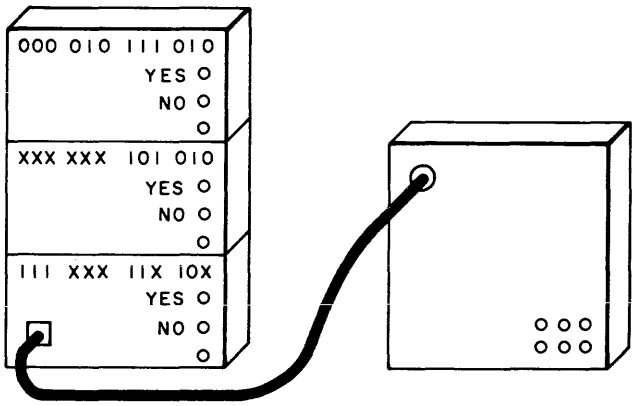


Figure 32 – Separate detector stack

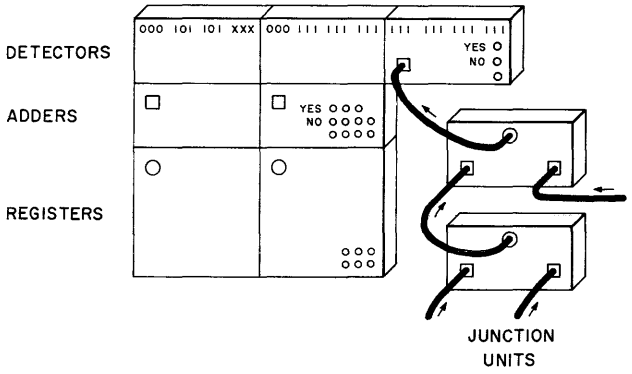


Figure 33 – Combined detectors

are used in an equivalent manner for detection of carry overflow, negativity, and the numerical sum zero. A signal returned at a "Yes" terminal indicates that the associated condition exists.

Detector units make it possible to select one of two alternative control paths on the basis of particular data values or patterns. Sometimes, however, it is desirable to select one of 2^n paths on the basis of n bits of data, and for such cases a decoder unit is provided (Figure 34). This unit contains a 3-bit decoder which may be interrogated by a control signal. Data input comes to the decoder either from the underlying cell or via a data cable. The input is passed upward to the overlying cell. Jumpers within a unit select three of the 12 data lines for decoding. When a control signal arrives at the interrogate terminal, a signal is produced on one of the eight output terminals. Each output terminal corresponds to one of the eight possible values encoded on the selected bits.

To permit decoding of values encoded on fewer bits, the jumpers for bit selection can provide an apparent "0" to the decoder. If, for example, the most significant bit of the decoded subset is thus fixed, an output signal will never appear on lines 4, 5, 6, or 7.

Figure 35 shows a stack of decoder units which splits the control path into one of 32 alternatives based upon bits 0-4 of the data input.

Merge unit

At some point after making a decision, all of the decision-dependent steps will have been executed, and the corresponding alternate control paths may be joined through the use of a merge element, several elements being housed in a merge unit (Figure 36). A merge element produces a signal (Z) at its output terminal whenever a signal appears at either input terminal (X or Y). Cascading permits the merging

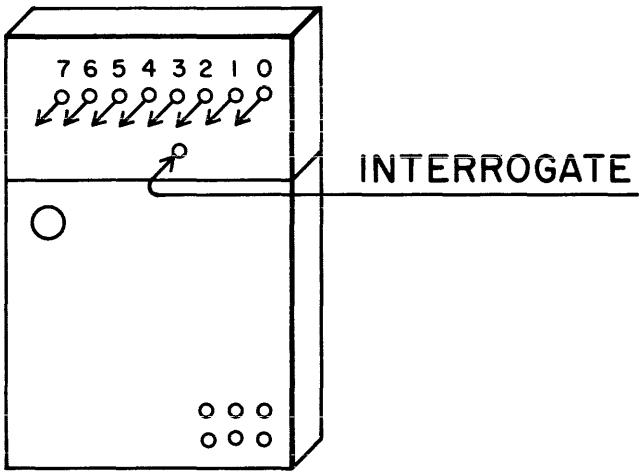


Figure 34 – A decoder unit on a register

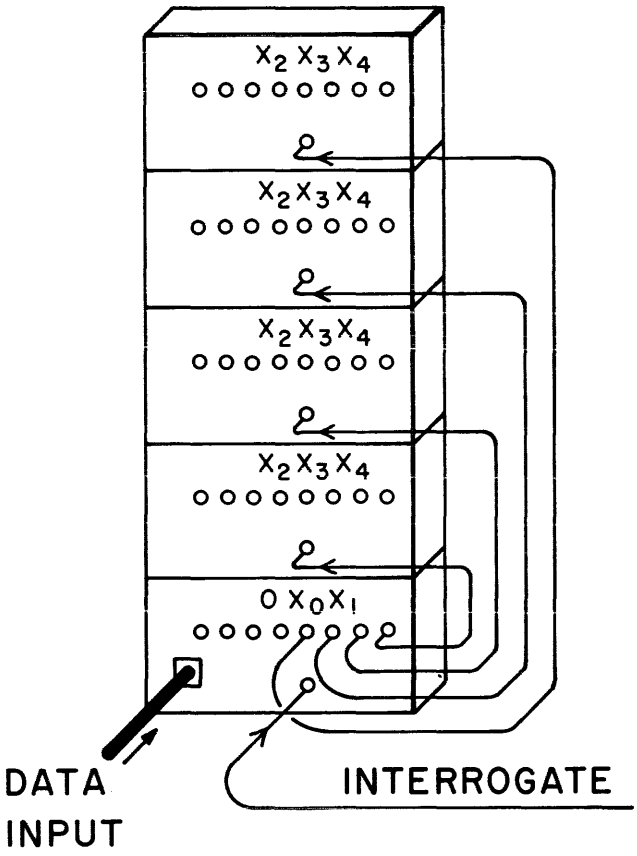


Figure 35 – Decoding five bits

of as many paths as desired. This unit, like the control branch, may also be used to interconnect, and thereby extend, control cables.

Decision call unit

A decision call unit permits a detector unit to be accessed by more than one control path and contains four decision calling elements as shown in Figure 37. This unit is connected via control cables to the control terminals of a detector unit, terminal 5 to the interrogate terminal and terminals 6 and 7 to the "No" and "Yes" terminals as shown in Figure 38. When a control signal is presented at the input of any of the four decision calling elements, a signal is produced at terminal 5 which interrogates the detector unit. A "Yes" or "No" signal is returned to the decision call unit and will appear at the "Yes" or "No" terminal of the element which called for the interrogation.

Decision call units can be cascaded (Figure 39) to allow an arbitrarily large number of control paths to access the same detector unit. Like the call unit, the decision call unit may be used to provide multiple access to a sub-sequence control path. In this case, the sub-sequence may include a decision in which one of two alternative control paths is selected.

Interlocking

In some situations two independent sequences will both require the use of the same data-processing element or elements (e.g., two sequences which make use of the same memory), and conflicts may arise. For such situations an interlock unit (Figure 40) is provided. This unit sorts incoming control signals on a "first-come, first-served" basis, interlocking them in such a way as to resolve conflicts.

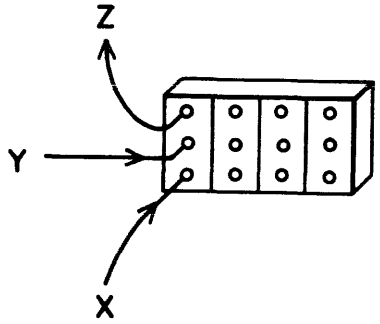


Figure 36 – Merge unit

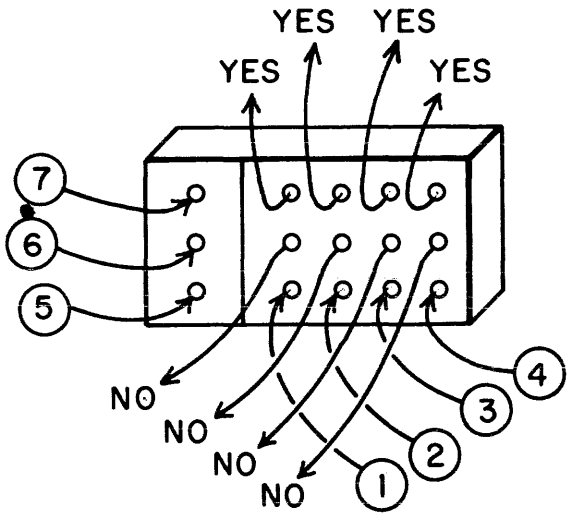


Figure 37 – Decision call unit

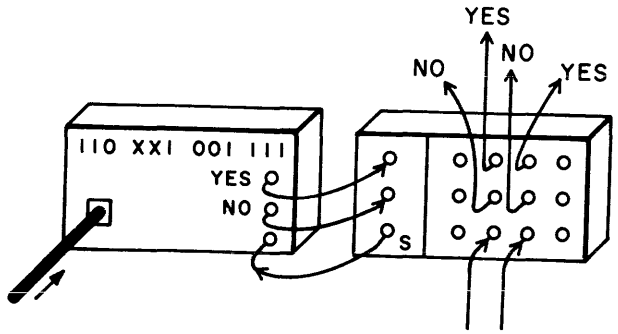


Figure 38 – Example of decision calling

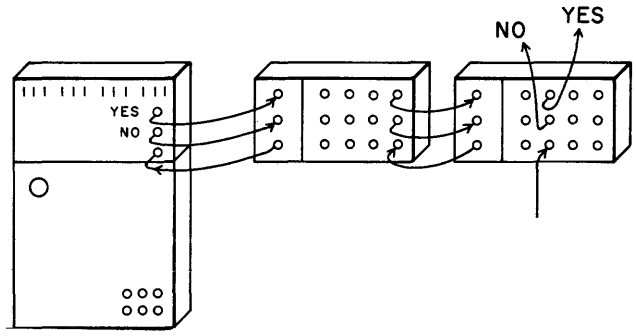


Figure 39 – Extending decision calls

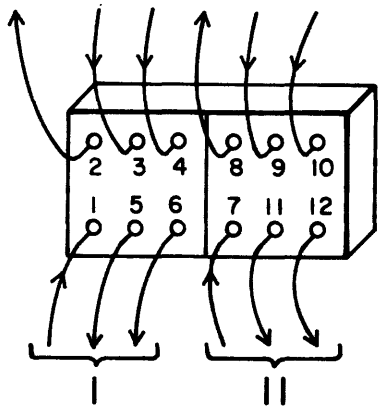


Figure 40 – Interlock unit

The left and right halves are associated with the control paths, I and II, of two concurrent sequences which must be interlocked. For a sequence to enter an interlocked phase, a signal must be presented to the interlock at an input terminal (terminal 1 for I, terminal 7 for II). Because it must be assumed that the sequences do not necessarily contain the same steps within their interlocked phases, each control path is provided with its own terminals (terminals 2, 3, 4 for I; terminals 8, 9, 10 for II) for use during their interlocked phase. If the interlock is off when a signal arrives at terminal 1, it is turned on and a signal is produced at terminal 2. This signal initiates the steps within the interlocked phase of the sequence associated with control path I. After the last of these steps has been completed, a signal is returned to terminal 3 or terminal 4. (Two return terminals are provided to allow for a possible decision within the interlocked section.) The return produces a signal at either terminal 5 or 6, depending on whether the return came to terminal 3 or 4, and shuts off the interlock. An equivalent process takes place for control path II, using terminals 7 through 12. If either control signal enters the interlock while it is on, it will be held up until the interlock is turned off. If signals arrive at terminals 1 and 7 simultaneously, only one will be accommodated immediately; the other will wait its turn.

Figure 41 shows an arrangement for interlocking two sequences (I & II), both of which use register A. Interlock units plugged into laterally adjacent cells (Figure 42) permit interlocking of any number of sequences.

Example of a small computer

Let us consider the central processing portion of a very simple computer and sketch out how it might be realized in macromodular form. The computer has a 12-bit word length, 4096 words of programmable memory, and an instruction repertoire consisting of

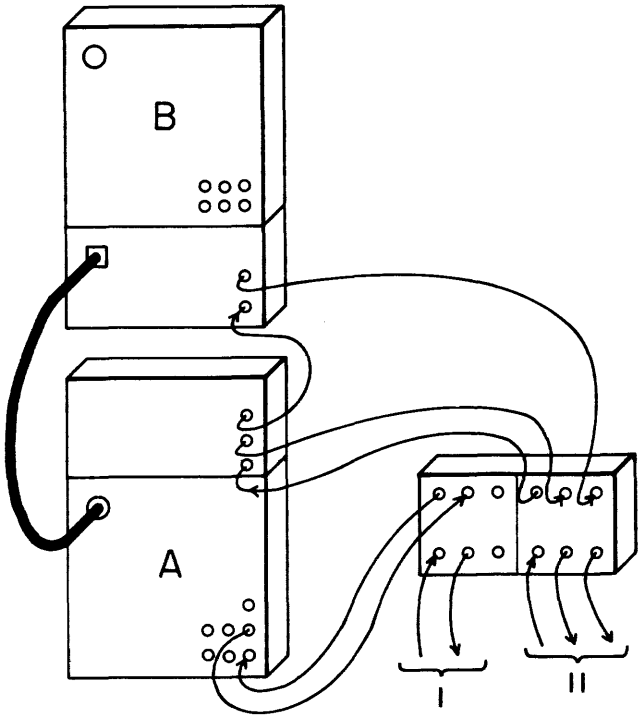


Figure 41 – Example of interlocking

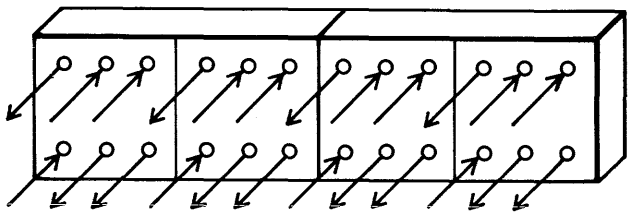


Figure 42 – Extended interlocking

eight instructions encoded on the leftmost three bits of the 12-bit instruction word.

Three of the instructions make reference to other memory locations by a process of indirect addressing. These instructions are ADDβ, STOβ, and JMPβ and use the contents of memory register $\beta(0 \leq \beta \leq 777_8)$ as the effective address. Thus, for example, if register β contains the number 1476, execution of an ADDβ instruction will cause the contents of register 1476 to be added to the contents of the accumulator. Similarly, a STOβ will store the contents of the accumulator in register 1476. A JMPβ will cause the next instruction to be taken from location 1476.

The remaining five instructions do not make reference to other memory locations. Instead, they perform the following functions:

- CLR - Clear the accumulator
- COM - Complement the contents of the accumulator.
- APO - Skip the next instruction if the accumulator contains a positive number.
- SHL n - Shift the contents of the accumulator n places to the left where n is specified by the four rightmost bits of the instruction word.
- NOP - Proceed immediately to the next instruction.

Figure 43 gives a two dimensional view of the processing network for this machine. Register A is the accumulator; register S is used to provide addresses to the memory and also to count out the number of shifts required by an SHL n instruction. During the execution of an instruction, register P holds the address of the next instruction to be executed. The decoder unit is set to decode the instruction field (3 most significant bits) of the word from memory, (M). The junction unit is set to mask out the instruction field when transferring β to S.

Figure 44 shows a flow diagram representation of the operation of the machine. Entering the top of the flow diagram corresponds to getting the next instruction from the memory. The contents of P are transferred to S in order to locate the instruction, and P is then indexed in preparation for locating the following instruction. Memory is read to obtain the instruction and the decoder is signaled to decode the instruction field of the memory word. Eight decoder output control paths are provided, one for each of the instructions, and a control signal is produced on the selected path. After the required sequence of steps has been executed, the control signal returns to get the next instruction, (GNI).

Figure 45 shows the control paths required to locate and decode the instructions, as well as the execution control paths for the instructions CLR, COM, APO, and NOP. The detector in the APO control path is set to detect a "0" in the leftmost (sign) position of the accumulator. A "No" signal response from the detector proceeds to the merge unit assemblage from which a signal to get the next instruction emerges. A "Yes" response indexes register P and then proceeds to get the next instruction. The NOP instruction has no execution steps, and its control path is routed directly to the GNI merge unit assembly.

The SHL n instruction, shown in Figure 46, uses the the S register to count the number of places which have been shifted. The SHL n instruction word is transferred from the memory into S and complemented. A detector on S monitors the rightmost four bits, (n), and provides a "Yes" response when the value 17_8 (the minus zero in ones' complement form) is detected. A control loop is entered in which the detector is interrogated. A "No" signal response from the detector shifts A one place, indexes the count in S, and again checks the detector. This process repeats as long as more shifts are required. When the required number of shifts has been completed, interrogation of the detector will produce a signal at its "Yes" output, which is routed to the GNI merge unit assemblage.

The control paths for the three memory reference instructions are routed to a sub-sequence all unit. This sub-sequence fetches the operand address and places it in the S register. This three-step sub-sequence is shown separately on the flow diagram in Figure 44 and the control path which realizes it is

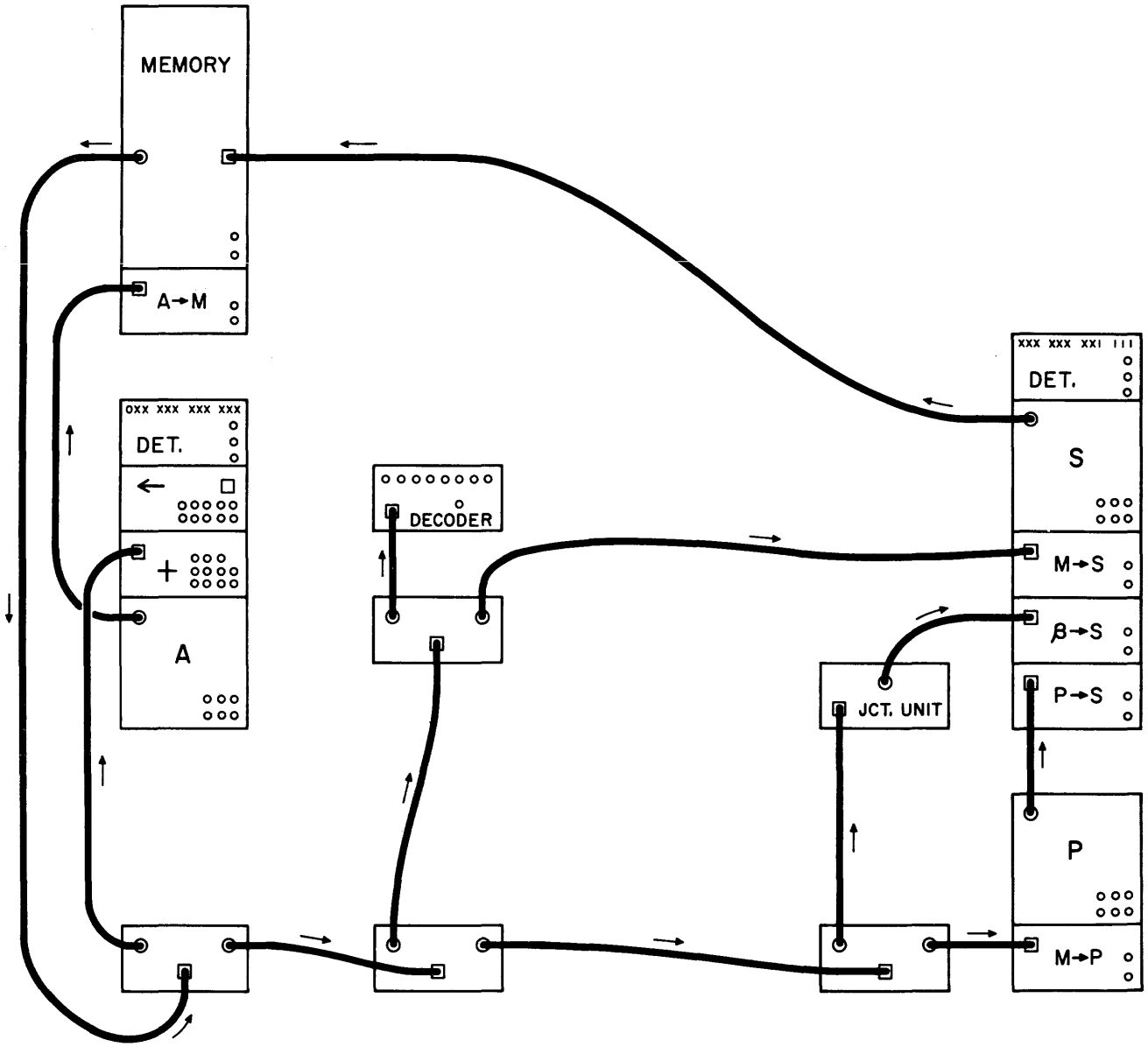


Figure 43 – Small computer processing network

shown in Figure 47. The first step of the sub-sequence is the transfer of β to S. In the following steps, the effective address is obtained from memory register β and transferred into S. Beyond this point the three instructions' control paths diverge.

In the ADD instruction the memory is again read to obtain the operand, which in the next step of the sequence, is added into A. The STO instruction execution sequence consists of a single Write command to the memory, as register A is connected directly to the memory data input. In the JMP instruction, the effective address is transferred into P from the memory. For this instruction the final step of the operand address fetch sub-sequence (which put the operand address into S) was unnecessary but harmless.

This example demonstrates the ease and directness with which simple systems can be put together. A total of 13 data cables, approximately 50 control cables, and 26 modules are required to form this complete, albeit comparatively modest, central processor.

CONCLUSION AND ACKNOWLEDGMENTS

The ideas presented here have evolved gradually through a combination of individual effort and group discussions. The authors wish to express their gratitude particularly to A. Anne, J. R. Cox, Y. H. Chuang,

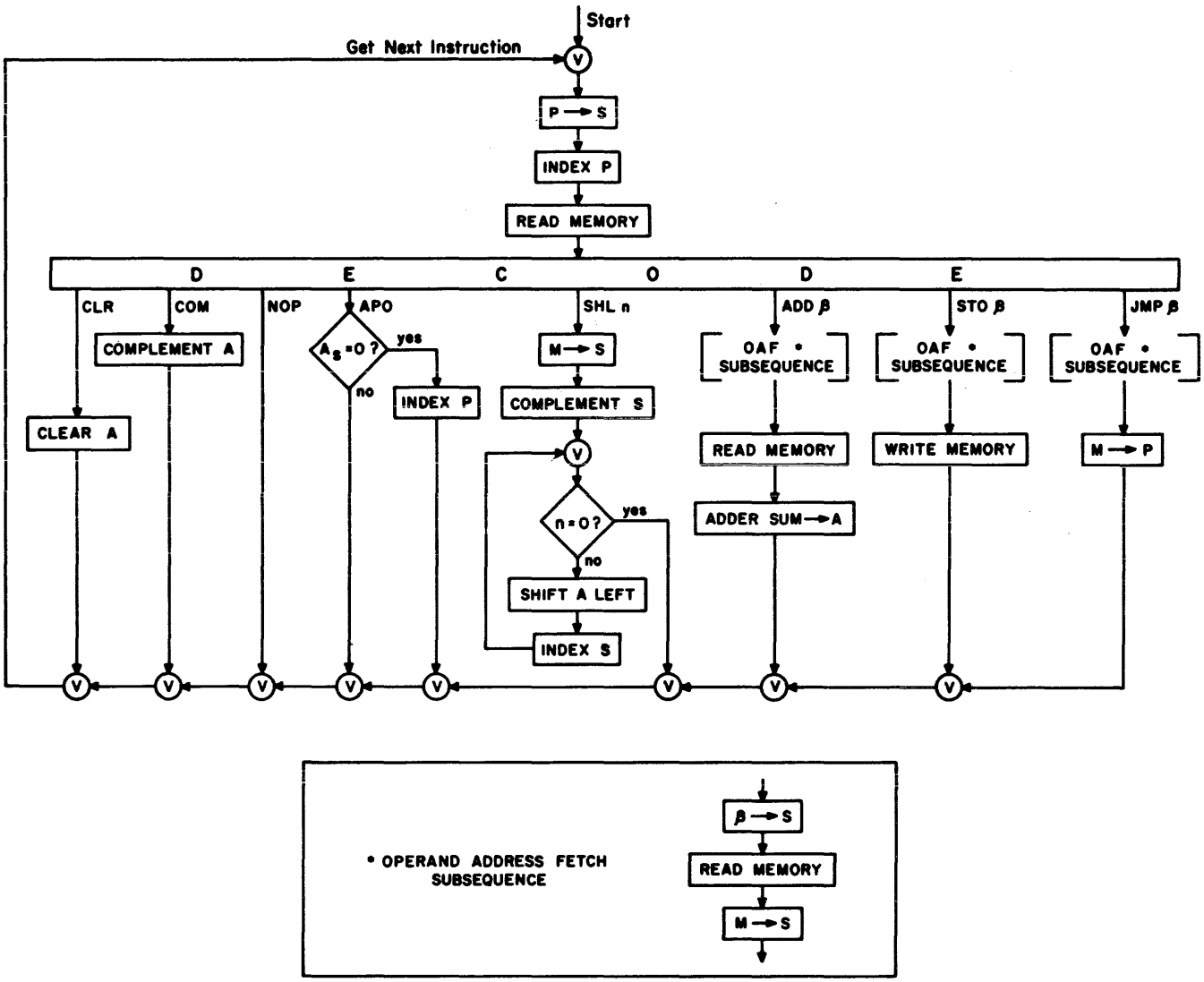


Figure 44 – Small computer control flow diagram

R. A. Ellis, G. C. Johns and C. E. Molnar who have contributed helpful criticism and suggestions.

While some details of design are still in flux, the die for the basic scheme is cast and an initial evaluation effort is under way. Prototypes of the macromodules are working and some small initial systems are planned for coming months. Paper design of these systems indicates that the particular functional breakdown we have chosen is a reasonable and convenient one.

Addition of new modules to the inventory together with some reshaping of those presently defined will certainly take place as experience guides us toward ever increasing convenience and flexibility.

REFERENCES

1 W A CLARK
Macromodular computer systems
Proc SJCC 1967

2 M J STUCKI S M ORNSTEIN and W A CLARK
Logical design of macromodules
Proc SJCC 1967

3 E G ANDREWS
The Bell computer model VI

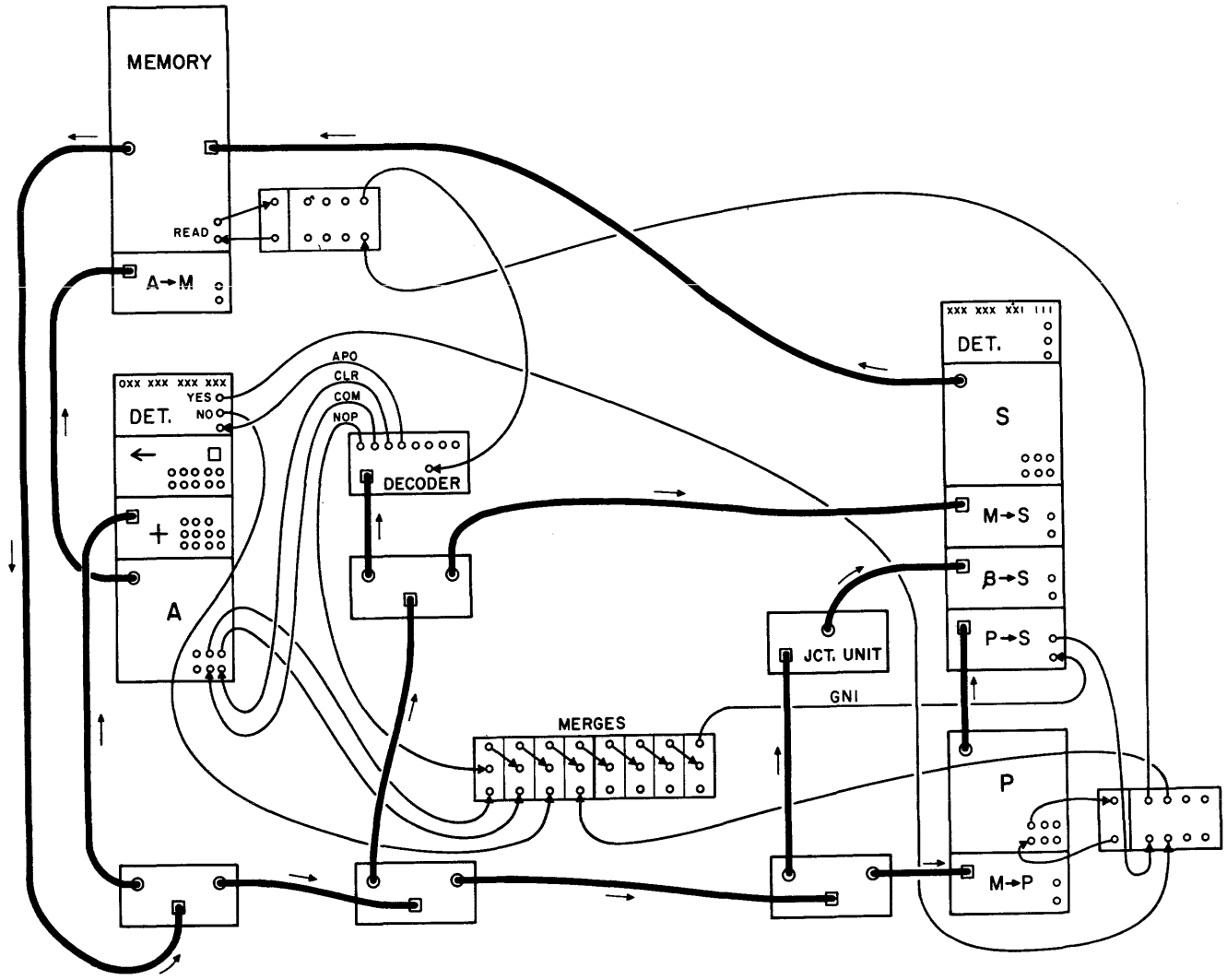


Figure 45 – Small computer-sequencing network for CLR(COM, APO, and NOP instructions

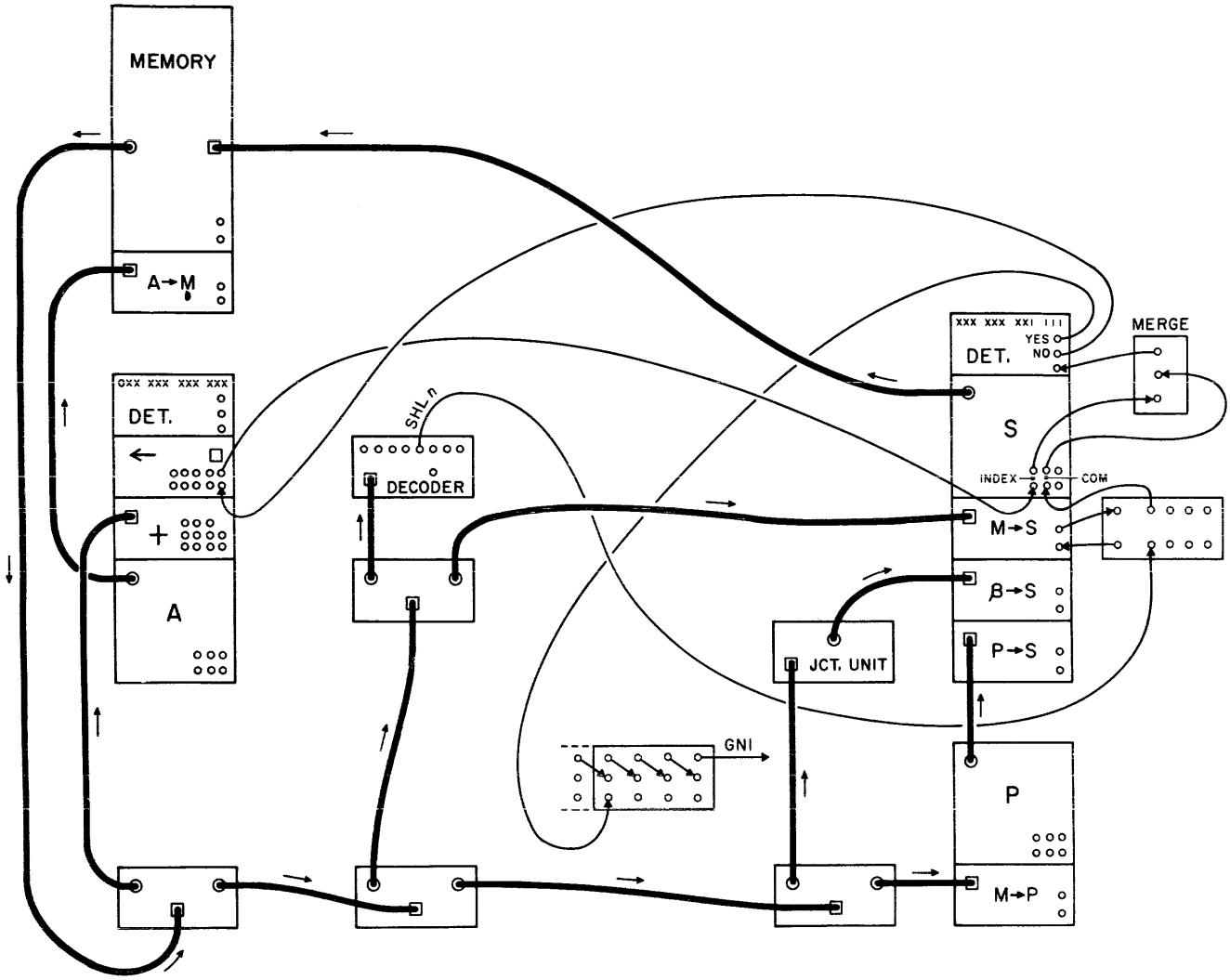


Figure 46—Small computer-sequencing network for SHL n instruction

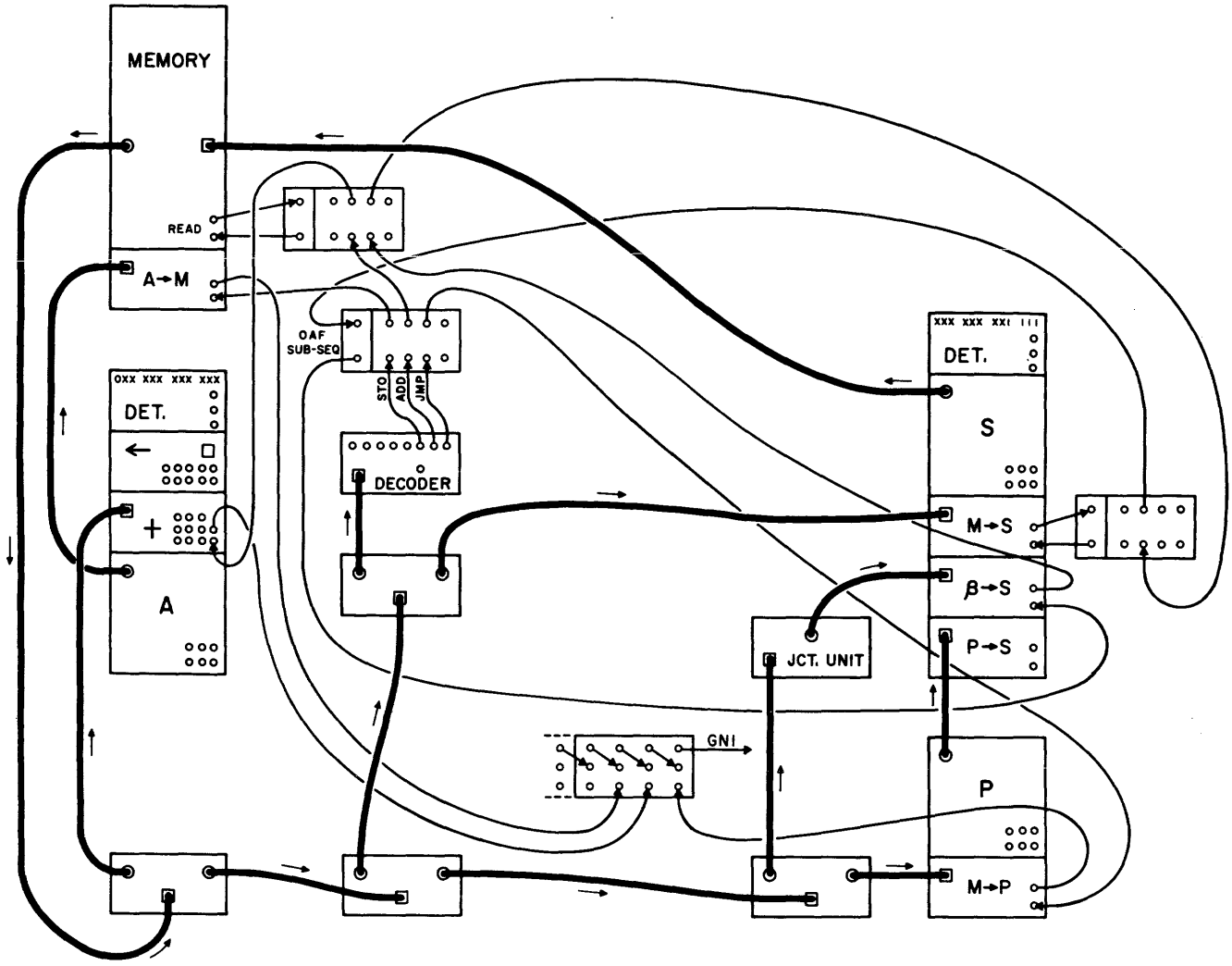


Figure 47—Small computer-sequencing network for ADD, STO, and JMP instructions

Logical design of macromodules*

by MISHELL J. STUCKI, SEVERO M. ORNSTEIN and WESLEY A. CLARK
Washington University
St. Louis, Missouri

INTRODUCTION

The macromodules[1,2] being developed at Washington University are logical building blocks which can be inserted into a special frame and inter-connected by standardized cables to form digital computing systems of any desired complexity. The logical design of these modules is fraught with many problems, some of which yield easily to standard design techniques and others which do not. The purpose of this paper is to present the design approaches in present use for the handling of problems of the latter type. Specifically, rather than present the details of adders, shifters, registers, etc., discussion is confined to those aspects of the logic within the modules which simplifies the job of assembling the modules into a working system. The general areas of asynchronous control, data validation, and word-length extension are discussed and design approaches presented. These approaches are then illustrated in the design of the macromodular data transfer operation, and the paper concludes with a few general comments on the circuitry now in use.

Asynchronous control

One of the more interesting aspects of the macromodules is the general control scheme which allows complex system control structures to be implemented with relative ease. In this scheme, data processing modules are designed for asynchronous control and special control modules are provided for the paralleling and conditional branching of control signals. By "asynchronous control" it is meant that associated with each data processing operation that a module can perform is an initiation terminal and a completion terminal; execution of an operation begins when a control signal arrives at the initiation terminal, and at the conclusion of the operation, a control signal is generated at the completion terminal. The design of circuitry exhibiting this kind of behavior is one of the more difficult logical design problems. The approach used is to partition the circuitry for a given data processing operation (such as addition) into less complex asynchronous circuits and to regulate their behavior with a control network that utilizes the basic circuits of the control macromodules. The basic control circuits and their usage in control networks are described in the following paragraphs.

The circuitry associated with an operation designed for asynchronous control is represented diagrammatically by a circle containing the name of the operation (Figure 1). The incident and extant arrows

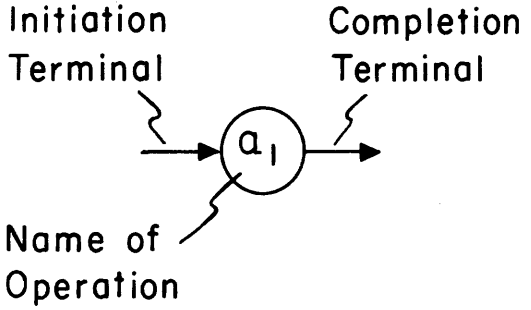


Figure 1 – Diagrammatic representation for the circuitry associated with an operation designed for asynchronous control

indicate connections to the initiation and completion terminals of the circuit. Operations can be made to occur in a specific sequence by connecting the completion terminal of each operation to the initiation terminal of the succeeding operation. Figure 2, for example, shows the flow diagram and connection network for the sequence a_1-a_2-a_3. The arrows in Figure 2b indicate that the completion terminal for a_1 is connected to the initiation terminal for a_2 and the completion terminal for a_2 is connected to the initiation terminal for a_3. Connected in this way, the completion signal from each operation initiates the next operation in the sequence.

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

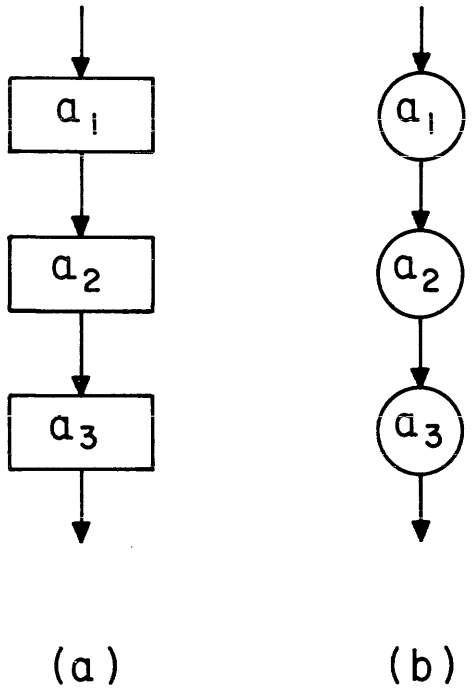


Figure 2—Flow diagram (a) and control network (b) for a simple sequence

Figure 3 shows the flow diagram and control network for a sequence containing a conditional branch: the diagram defines the sequence a_1-a_3-a_4 when X=0 and the sequence a_1-a_2-a_4 when X=1. X is assumed to be a binary variable. The diamond-shaped element in Figure 3b is connected to the completion terminal for a_1 and the initiation terminals for a_2 and a_3. This element, called a decision (D) element, is a circuit that routes the completion signal coming from a_1 to the initiation terminal for a_2 or a_3 depending on the value of X.* The M element in Figure 3b is connected to the initiation terminal for a_4 and the completion terminals for a_2 and a_3. This element, called a merge (M) element, is a circuit that routes the completion signal coming from a_2 or a_3 to the initiation terminal for a_4.

Figure 4 shows the flow diagram and control network for a sequence in which operations a_2 and a_3 are to be executed in parallel. In the control network, the completion terminal for a_1 is connected to the initiation terminals for a_2 and a_3 so that the completion signal from a_1 will initiate both operations. The (R) element in the network is connected to the initiation terminal for a_4 and to the completion terminals for a_2 and a_3. This element, called a rendezvous (R) element, is a circuit that generates a control signal as soon as it has received a completion signal from both a_2 and a_3. Inclusion of the R element in the network guar-

*Variable X is supplied to the D element at a terminal not shown in the figure.

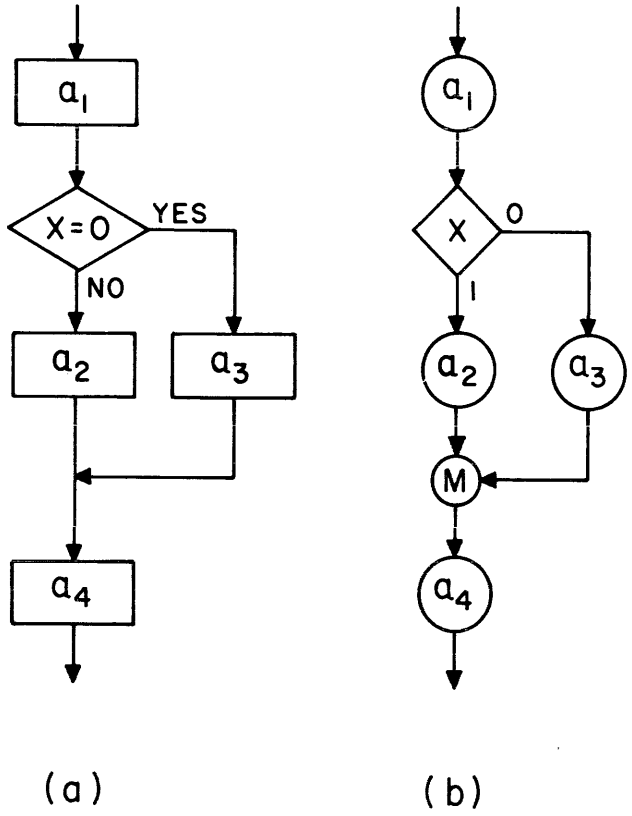


Figure 3—Flow diagram (a) and control network (b) for a sequence containing a conditional branch

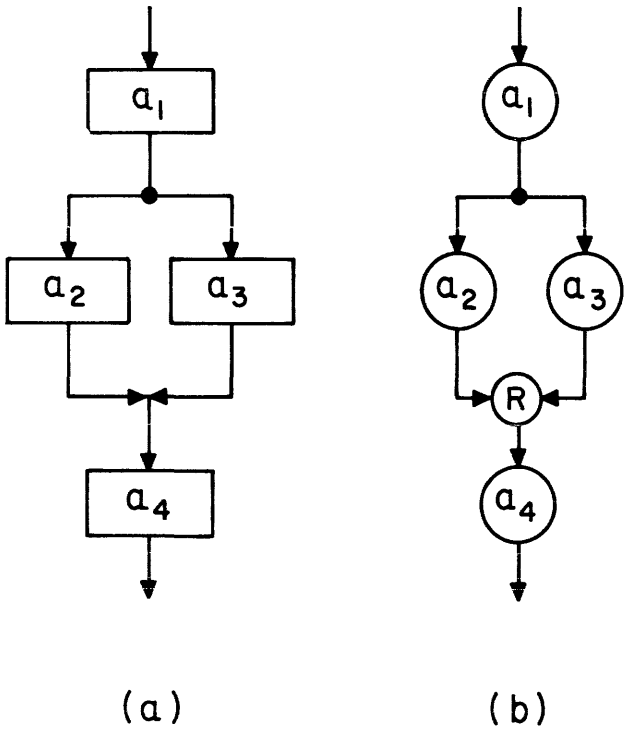


Figure 4—Flow diagram (a) and control network (b) for a sequence involving parallel execution of operations

antees that operations a_2 and a_3 will both be completed before operation a_4 is initiated.**

The correspondence between a flow diagram and a control network is not always one-to-one as implied by the preceding examples. The difference arises when an operation occurs in several sequences or in more than one place in the same sequence. In either case, independent access to the control terminals for the operation is required at several points in the network, and this cannot be accomplished by connections made directly to the control terminals. Consider, for example, independent sequences a_1-a_3-a_4 and a_2-a_3-a_5. Both require operation a_3, and direct connection to the control terminals of a_3 would result in the network shown in Fig. 5a. It is easy to see that this network does not describe two independent sequences. Proper implementation of the sequences, the network shown in Fig. 5b, uses a call (C) element to keep the two sequences separate. The C element has three pairs of control terminals, one of which connects to the control terminals of the operation to be multiply accessed. The other two pairs, indicated by the dotted lines, act as independent control terminals for the operation. When the completion signal from a_1 arrives at the C element, the element initiates operation a_3 and routes the completion signal from a_3 to the initiation terminal of a_4. When a completion signal from a_2 arrives, the C element initiates operation a_3 and routes the completion signal from a_3 to the initiation terminal of a_5.

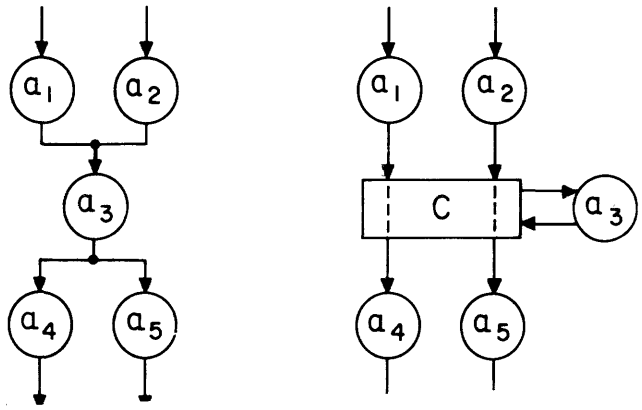


Figure 5 – An improper way (a) and a proper way (b) of incorporating an operation's circuitry in two sections of control network

**It must be noted that if one of the operations is known to have a longer execution time than the other, its completion signal may be used to initiate a_4 and the R element can be removed from the network. It is assumed in this paper that the execution time of an operation is unknown, and an R element will therefore always be used to terminate parallel processes.

Data validation

Before any data processing operation may be executed, the latest value of the data to be used must be guaranteed present at the circuitry associated with the operation. This is a severe problem in macromodular systems since the physical separation between modules is not constrained and data propagation times are therefore unknown and unbounded. The problem is solved by a scheme called Data Validation (DV): when a data processing operation disturbs a source of data, the module does not generate a completion signal for the operation until the new data value has propagated to all parts of the system that may need it. Since subsequent data processing operations that use the data cannot occur until the completion signal is generated, the new data value is guaranteed available at the circuitry for these operations.

Data is carried from one module to another by means of special data cables or by inter-cell connections in the system frame. Included in each type of data path is an extra pair of lines carrying control signals associated with the DV process (Figure 6). Whenever a new data value begins to propagate along a data path, the data source transmits a control signal along line I of the path. This signal is generated slightly later than the propagating data value, and the data path and associated circuitry are designed so that the signal consistently lags the data. Hence, when the control signal reaches the other end of the data path, the circuitry there is guaranteed that the new data value has also arrived. As soon as this circuitry has assimilated the new data value, it sends a control signal back to the data source via line C of the path.

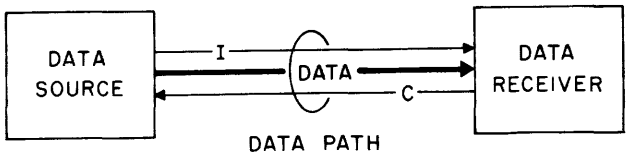


Figure 6 – General structure of a data path

The I and C lines of a data path can be treated as control lines for an operation called dv. Figure 7a, for example, shows part of the internal control network of a module at the receiving end of a data path. The control lines labeled dv are the I and C lines of the data path, and operation a_1 is the operation (if any) performed by the module when notified of the arrival of a new data value. Figure 7b shows part of the internal control network of a module at the source end of a data path. It is assumed that operations a_1 and a_2 disturb the data source and the DV process is therefore performed after each of the

operations. Figure 7c shows the same module designed for two data paths. The DV process for one of the paths is called dv_1 and the DV process for the other path, dv_2. As shown, dv_1 and dv_2 are executed in parallel.

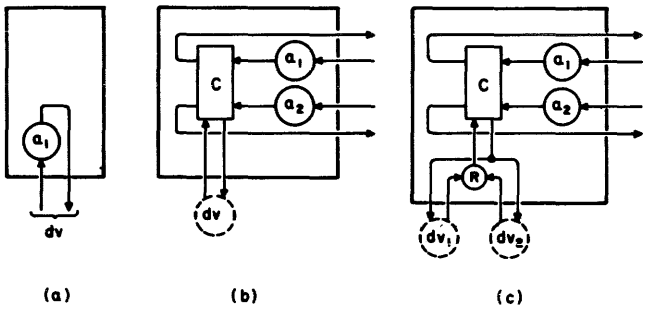


Figure 7 – Examples showing the incorporation of the DV process in the internal control networks of modules at either end of a data path

Wordlength extension

In order to simplify the control structure that a user of macromodules has to construct, system control has been made independent of data wordlength. Data processing modules are organized in parallel form with a wordlength modulus of 12 bits; a register module, for example, contains a 12 bit register and the adder module contains a 12 bit adder. If the user requires a wordlength of more than 12 bits, data processing modules can be cascaded to handle greater wordlengths by plugging them into laterally adjacent cells in the system frame. Inter-cell connections within the frame allow the internal control networks of the individual modules to link together to form a single control network for the entire array. This network, called the Wordlength Extension (WE) network, relegates operational control of the array to the control terminals on the rightmost module so that a data processing operation has but a single pair of control terminals regardless of the wordlength involved.

That portion of a WE network contained within a single module is called a WE segment, and the general form of the WE segment for a single data processing operation is shown in Figure 8. The terminals on the right are the control terminals associated with the execution of the operation in this module; the terminals on the left connect to the control terminals on the module in the left-adjacent cell that are associated with the execution of the operation in that module. The segment is designed according to the following rule: if it sends an initiation signal to the module in the left-adjacent cell, it must receive a completion signal from that module before it can generate a completion signal of its own. This rule guarantees that a segment will not generate a completion signal until all activated segments to its left have generated completion signals; hence, the completion signal generated by the rightmost segment is the completion signal for the entire array.

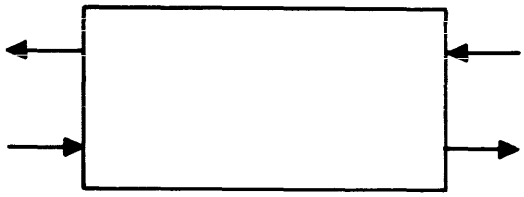


Figure 8 – Generalized representation of a WE segment

WE segments are composed of subsegments as shown in Figure 9. The segment labeled BD is a boundary delimiting network; its purpose is to inhibit communication with the left-adjacent cell if that cell does not belong to the array. The flow diagram for the BD segment is shown in Figure 10 and is the same for all WE segments. The segment labeled DP is that portion of the WE segment which carries out the data processing operation in the module.

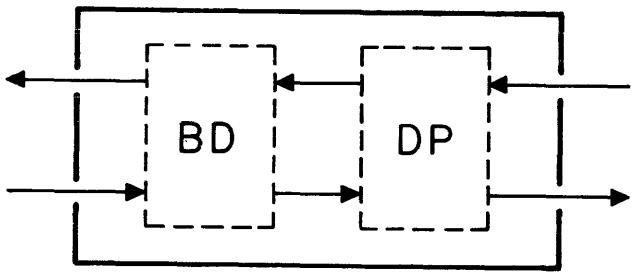


Figure 9 – A WE segment designed in terms of subsegments.

Fig. 10 and is the same for all WE segments. The segment labeled DP is that portion of the WE segment which carries out the data processing operation in the module.

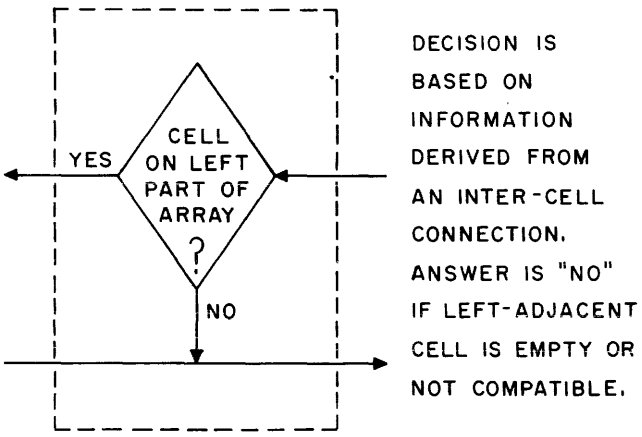


Figure 10 – Flow diagram of the BD subsegment

Figure 11a, for example, shows the DP network for the operation CLR (set all bits to zero). The network here is a parallel one, a control signal being

sent to the left at the same time that the CLR operation for the module is initiated. The R element guarantees that the network does not generate a completion signal until it has received a signal from the left. Figure 11b shows a serial network for the same

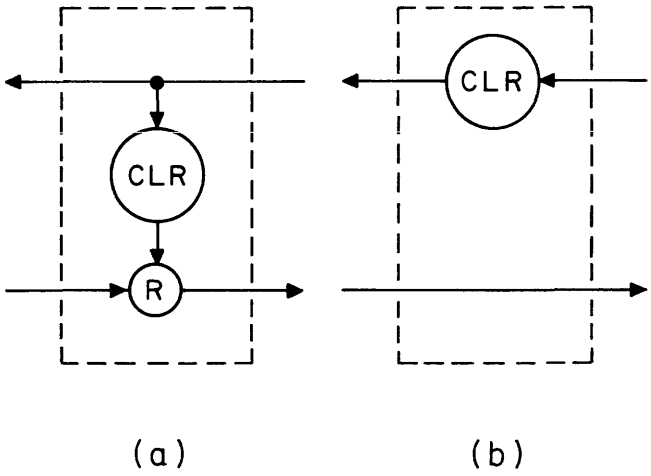


Figure 11 – A parallel network (a) and a serial network (b) for the DP subsegment associated with the operation CLR

operation. In this network, a signal is sent to the left only after the CLR operation for the module is completed; hence the network can generate a completion signal as soon as it receives a signal from the left. Figure 12 shows the flow diagram of a DP

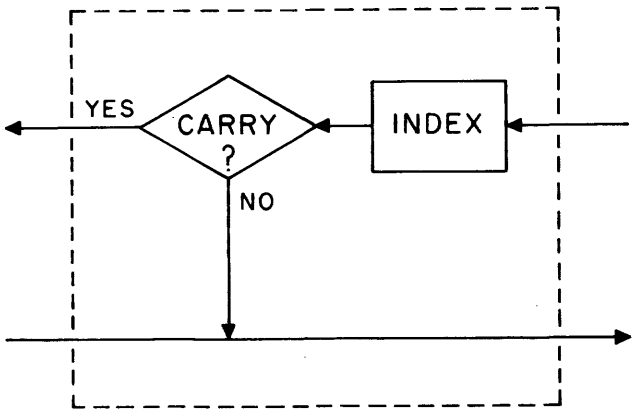


Figure 12 – Flow diagram of a DP subsegment for the operation INDEX

segment for the operation INDEX (add 1 to the value of the data). In this segment, a signal is sent to the left only if the INDEX operation in this module indicates a carry into the next module.*

*The most significant bit is in the leftmost module of the array.

Transfer of data

The transfer of data into a register is accomplished by means of a module called a data gate Figure 13). The terminals on this module are the control terminals for the transfer operation, and when the operation is initiated, the data on the lateral data path is transferred into the register. Data gates may be cascaded as shown in Figure 14 so as to allow data from any number of sources to be transferred into a register. The data paths running between the data gate modules allow each of these modules to communicate with the register.

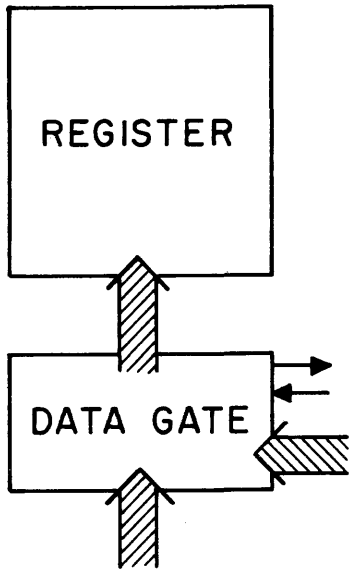


Figure 13 – Proper interconnection of a data gate module and a register module. Broad arrows represent data paths

The logic associated with the data path input of the register is shown in Figure 15. As indicated, the dv operation for the path causes the newly arrived data value to be loaded into the register. The general transfer process therefore consists of two steps: (1) the data to be transferred must be gated onto the data path input of the register, and (2) the dv operation for the path must then be initiated. These functions are provided by the data gate module at the other end of the data path. When a signal arrives at the data gate's initiation terminal (Figure 16), a flip-flop G is set (1→G) which gates the data of the lateral input path onto the output path, and the dv operation is then initiated. At the conclusion of the dv operation, the flip-flop is reset (0→G) so that the lateral path is no longer connected to the output path, and the module then generates a completion signal. When the transfer process is not being executed by the module, the output path is connected to the bottom input path shown in Figure 16 so that modules located

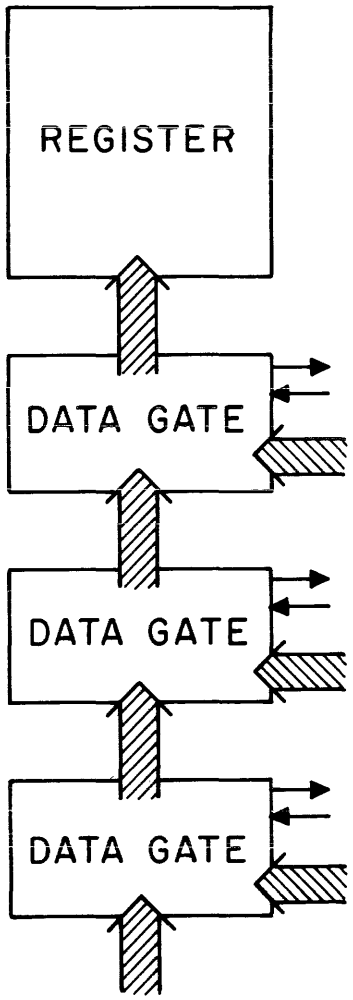


Figure 14—Data gate modules cascaded to allow data from three different sources to be transferred into a register module

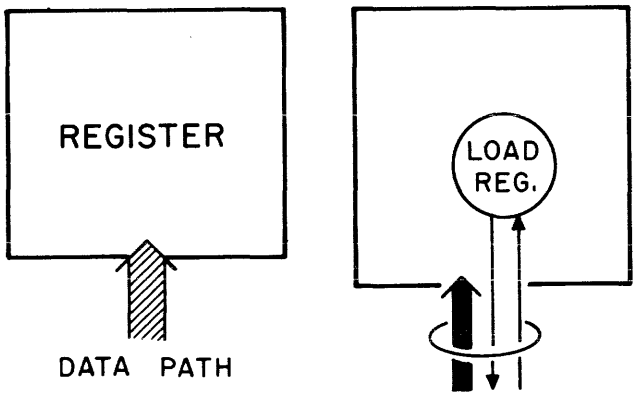


Figure 15—Register logic associated with the transfer process

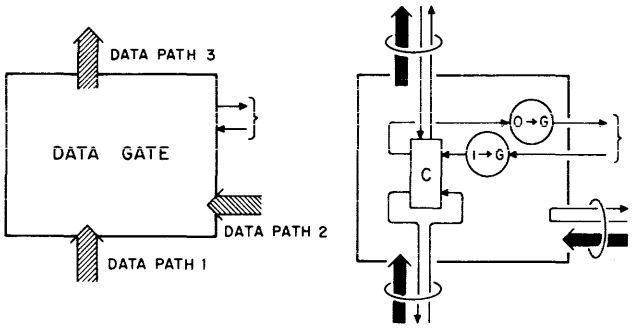


Figure 16—Data gate logic associated with the transfer process

below this module may communicate with the register. The dv process associated with the bottom input path initiates the dv process for the output path, thereby insuring that the dv process initiated by a lower module will propagate through this module and on to the register. The C element in the figure keeps the transfer sequence initiated by this module separate from transfer sequences initiated by lower modules.

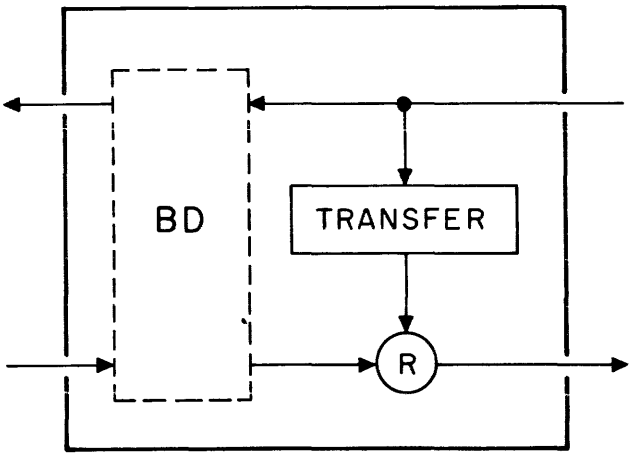


Figure 17—Flow diagram of the WE segment of a data gate module

Wordlengths of more than 12 bits are handled by laterally cascading data gate modules as described in the section on Wordlength Extension. The WE segment for a data gate is shown in Figure 17, and Figure 18 shows the assemblage of modules required for 36 bit transfers from three different sources.

Implementation

In present macromodular design, an initiation terminal accepts a binary input, a completion terminal produces a binary output, and a control signal is a change in value, either from 1 to 0 or from 0 to 1. The control elements are realized in asynchronous fundamental mode level logic form, and the state and output tables for each are given in Figure 19.

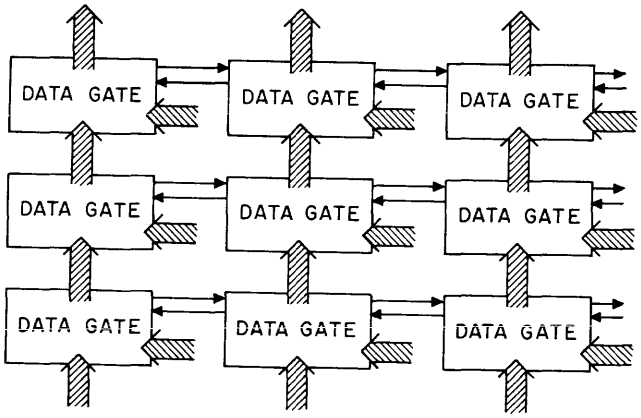


Figure 18—The assemblage of data gate modules required for transferring 36 bit words from three different data sources

CKNOWLEDGMENT

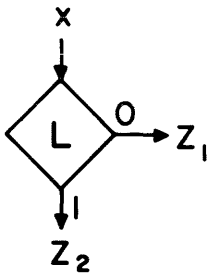
The authors wish to express their gratitude to A.Anne and Y.H. Chuang for their help in the development of the ideas presented here.

REFERENCES

1 W A CLARK
Macromodular computer systems Proc SJCC 1967

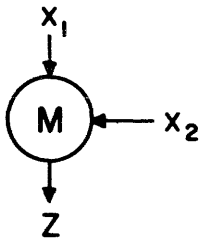
2 S M ORNSTEIN M J STUCKI W A CLARK
A functional description of macromodules
Proc SJCC 1967

(a) DECISION ELEMENT



	Lx 00	01	11	10	$Z_1 Z_2$
A	(A)	D	B	(A)	0 0
B	C	(B)	(B)	A	0 1
C	(C)	B	D	(C)	1 1
D	A	(D)	(D)	C	1 0

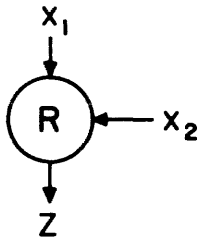
(b) MERGE ELEMENT



X_1 \ X_2	0	1
0	0	1
1	1	0

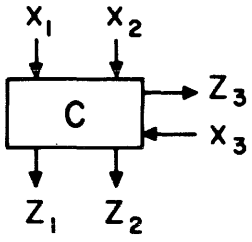
Z

(c) RENDEZVOUS ELEMENT



	$X_1 X_2$ 00	01	11	10	Z
A	(A)	(A)	B	(A)	0
B	A	(B)	(B)	(B)	1

(d) CALL ELEMENT



X_1 \ X_2	0	1
0	0	1
1	1	0

Z_3

	$X_3=0$ $X_1 X_2$ 00	01	11	10	$X_3=1$ 00	01	11	10	$Z_1 Z_2$
A	(A)	(A)	–	(A)	(A)	B	–	D	0 0
B	A	(B)	C	–	(B)	(B)	(B)	–	0 1
C	–	(C)	(C)	(C)	–	B	(C)	D	1 1
D	A	–	C	(D)	(D)	–	(D)	(D)	1 0

Figure 19 – State and output tables for the control elements

Engineering design of macromodules*

by ASHER BLUM, THOMAS J. CHANEY
and RICHARD E. OLSON
Washington University
St. Louis, Missouri

INTRODUCTION

Macromodules are intended for assembly into a wide variety of computer structures with as few restrains on the designers' choice of the geometry of the machine as it is practical to require. The goal of geometric flexibility makes the problem of reliably interconnecting the modules to communicate date more difficult than usual. The ad-hoc repair of transmission failures is precluded because such problems would simple relocate as the system structure is altered by its users, and the detection and repair of the faults would be necessary after each alteration. Thus, as a practical consideration, hardware designs which do not eliminate such failures limit the system flexibility. Consequently, a substantial proportion of the engineering effort has focused on the data communication problem. Because the power distribution system interconnects the modules, it represents a set of pathways over which undesired signals may propagate.

Therefore, power distribution has been considered in relation to the communication problem as well as in relation to its more direct effects on the design of the hardware. Because many details of the system and hardware design remain to be specified and because of the complexity of useful systems, no detailed noise calculations on such systems have been undertaken. Instead where a choice is permitted, the most conservative designs have been adopted with some restraints with respect to complexity. When the first small systems are complete, these early decisions will be re-evaluated, and, hopefully, at that point less conservative and simpler hardware will replace the present designs.

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218

A brief description of some typical characteristics of the modules helps in visualizing the design problems. As presently conceived, the macromodules will be selected from a set of 20 to 40 basic designs, and the bulk of the designs will range in logical complexity from as few as 50 to as many as 400 gate circuits. Current designs make use of emitter coupled integrated circuits having a typical rise time of six nanoseconds. It is anticipated that higher speed circuitry will be available in the near future and the hardware design is oriented toward the easy incorporation of this higher speed logic as it becomes available. The circuitry of a data gate module is shown in Figure 1 as a representative example of one of the smaller units.

Architecture

Two schemes of assembling the modules into a computing machine are under consideration. In the first scheme, the module cases mechanically interlock to form a rigid structure which is to be the whole of, or some portion of, the computer. The module case carries suitable sliding connectors which would permit adjacent modules to exchange digital information through the abutting sides. The front face of the module supports cable connectors through which information exchanged between non-adjacent modules travels, or for passing information between adjacent modules by means other than the sliding connector. Ground and power are distributed to the modules by a system of cables. A very simple system having these properties is shown in Figure 2. In the second scheme the modules are first mounted into a frame which has capacity for approximately 50 modules. The electrical paths provided by the sliding connector described above are replaced by a fixed array of wired interconnections between adjacent ports in

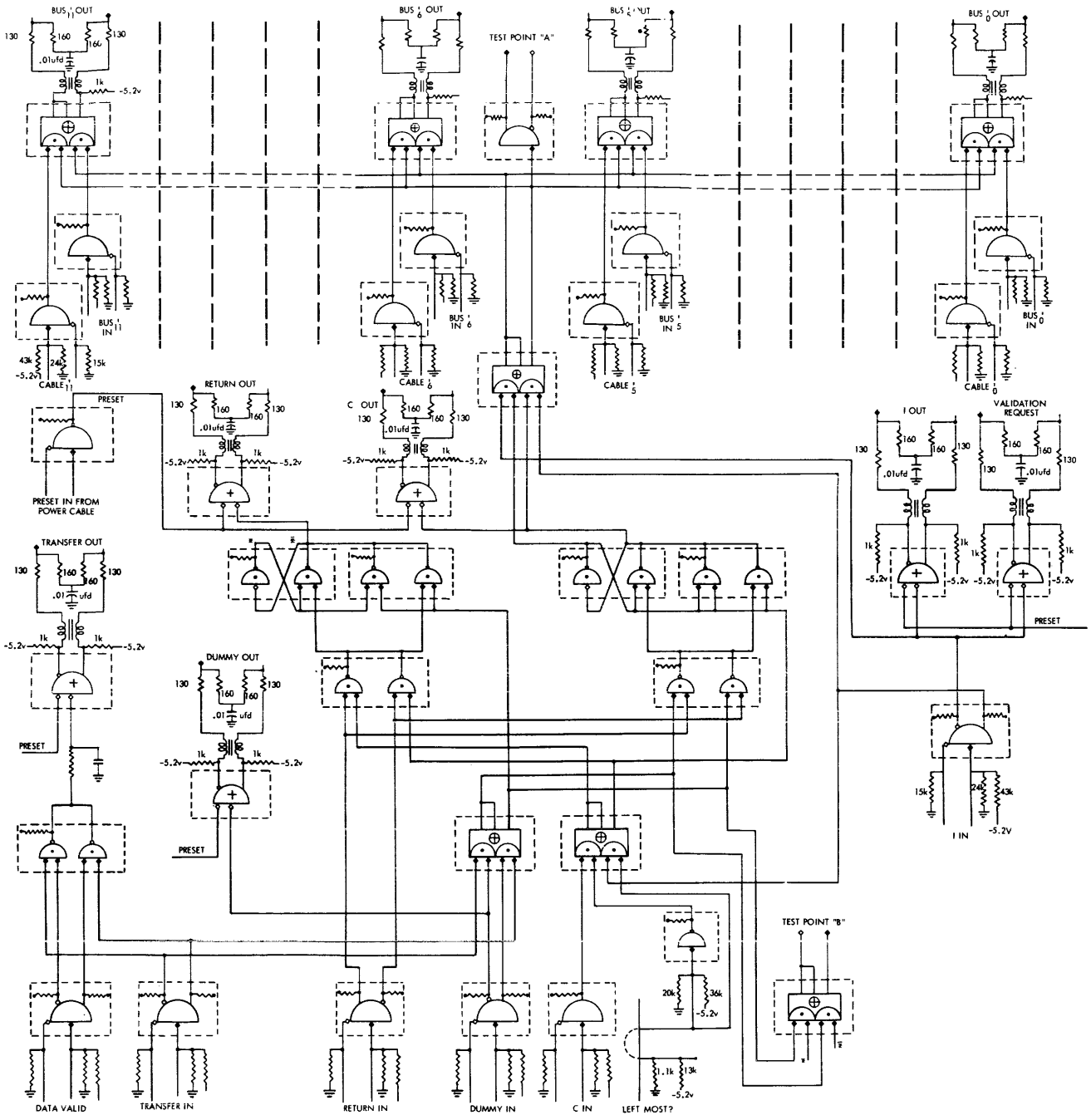


Figure 1 – A data gate circuit, illustrating the complexity of one of the simpler macromodules

the frame. The front face of the module case supports a bank of connectors, and somewhat over half of the pins in the connector bank engage with connectors which are fixed to the frame and on which the wiring mentioned above terminates for the purpose of providing data paths between adjacent modules. The remaining pins on the module face engage the connectors of a "face plate." This plate, one of which is associated with each module, carries the cable connectors which permit communication between non-adjacent modules, and connects mechanically to the frame rather than to the module. Cabled data is funnelled back from the faceplate through pin connections between the module and the faceplate, into the module. The modules may be removed from the frame without disturbing the faceplate, and in this way the electronics may be removed without affecting the wired-in operational structure of the computer. An exploded figurative view of one frame section appears in Figure 3. Power and ground are distributed

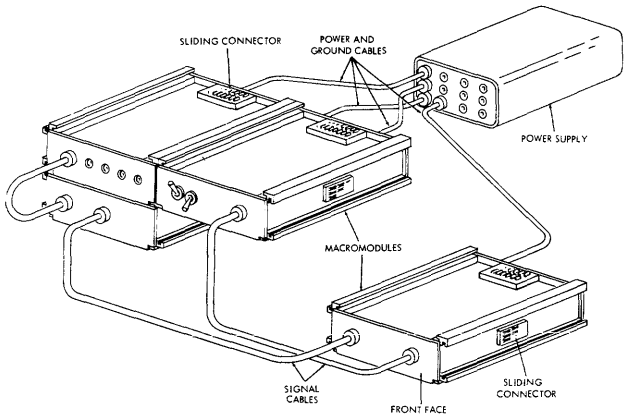


Figure 2—A simple system using mechanically interlocking macromodules

on the back of the frame, with power provided by a supply, not shown in the figure, which is attached to the frame. A computer is likely to consist of several such frames, and although one need not do so, one may bolt a number of frames together to form a single larger frame. A figurative drawing of such a computer is shown in Figure 4. (For the purpose of simplifying the drawing, the capacity of the frame has been reduced to twelve modules.)

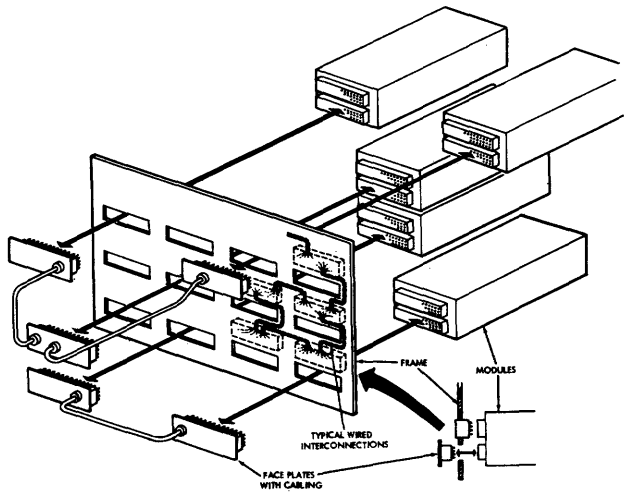


Figure 3—An exploded view of the frame, the associated modules and the faceplates

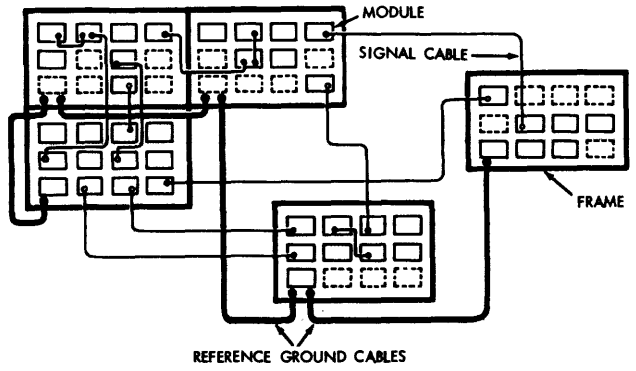


Figure 4—A system of macromodules mounted in frames

Data transmission problems

The electrical interconnections which carry digital information do so by carrying one of two d.c. levels separated by 0.8 volts. This low voltage swing together with a machine structure consisting of widely separated module assemblies cabled together and not enclosed in a single cabinet, suggest that unless unusual precautions are taken, noise will be a severe problem. The desirable mechanical flexibility conflicts with the compact, shielded type of structure which would be desirable from the point of view of noise. In the two modes of assembly described above, the noise problems differ only in detail. One may think of a frame holding modules as a module of increased complexity if, as is the case, the power and ground are distributed on a low impedance network.

Noise may be divided according to whether it is internally or externally generated. The externally generated noise is principally due to the electric and magnetic fields generated by equipment in the vicinity of the computer.

To attenuate the effects of external fields, a shielding system has been constructed around the computing machine, and this shielding reconfigures as the machine shape is changed. The shield system consists of the metallic module cases, and shielding which surrounds the signal leads, power cables, and ground cables. In the case of the frame supported system, the module case and the frame are electrically connected, so that the frame becomes part of the shielding. Where a lead or cable enter a module, the shield is terminated on the module case at the point of entry. The logic circuits are not grounded to their surrounding module cases. Thus, the shielding and cases form a low impedance path surrounding the whole system, but not electrically connected to it. It is anticipated that the shorted turns and low impedance paths to ground provided by the shielding will attenuate the noise generating effects of external electric and magnetic fields. The shielding of a small system is illustrated in Figure 6.

Among the sources of internal noise are the current and voltage transients which accompany a change in signal at some point inside the system. This effect may be reduced by transmitting signals over twisted-wire-pairs, shielded as described above, driven in a balanced mode. The twisted-wire-pair balanced mode couples very poorly with the surroundings. As a consequence, both the pickup and the radiation of noise in this mode are low. The integrated circuit logic provides, in many of the available gate packages, complementary outputs which are a suitable balanced signal source, If at the receiving end, one of the leads of the twisted-wire-pair is connected to the bias volt-

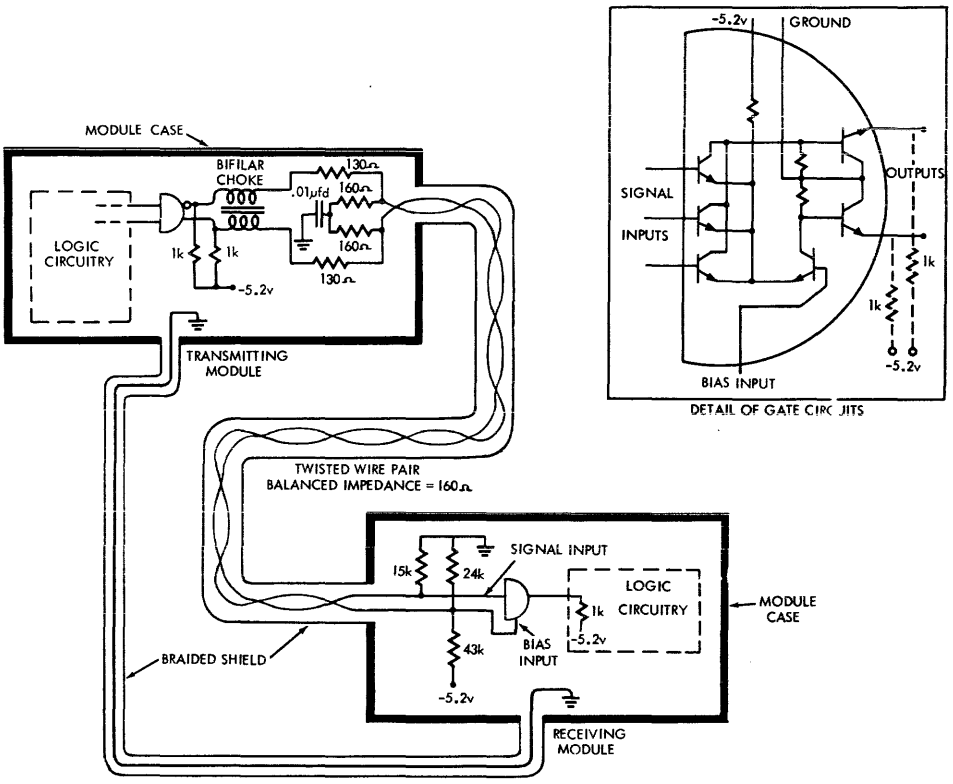


Figure 5 – Data transmission circuitry

age input, the receiving circuit will behave as a differential amplifier. The sensitivity to balanced signals will be undiminished, but the ability to discriminate against unbalanced mode noise will be raised substantially. Since the unbalanced mode is reasonably well coupled to the surroundings, almost all noise appears in this mode. The above arrangement increases noise immunity in this mode from the 0.2 volts of the typical unbalanced arrangement to over 1.0 volt.

A typical signal transmitting channel is shown in Figure 5. The output stage consists of an emitter coupled integrated circuit gate with complementary outputs; a bifilar, 100 μh choke; a .01 μfd capacitor; and several resistors. This network serves three purposes: (1) it provides a terminating impedance (at the driving end); (2) it acts as a voltage divider for the output signal, and (3) it provides a high output impedance to unbalanced currents. The 1K ohm resistors are provided to allow the output to swing negative when the emitter follower output is cut off. The detailed view of the output gate illustrates the need for these 1K ohm resistors. The choke is made by cementing two wires in parallel and winding the pair on a low Q ferrite core. Balanced currents will have cancelling fields and couple poorly to the core. Thus, the choke will represent a low impedance to these currents. The cancellation will not occur in the case of unbalanced currents. The choke will represent a high impedance to unbalanced currents and thus prevent their flowing out of the module to disturb the rest of the system. The output impedance of the integrated circuit is about 24Ω and forms a voltage divider when combined with the 130Ω, and 160Ω resistors. Although we have divided our driving voltage in half, the voltage differential at the receiving end is the same as in normal circuit usage. This is because the bias voltage input is not fixed as in the usual manner of use, but instead is driven oppositely to the signal input. The receiving gate acts as a differential amplifier. The unbalanced noise margin is increased by attenuating the balanced signal voltages as shown. The three resistors at the receiving end serve two purposes: (1) they bias the gate to a known state when the cable is not present; (2) they provide currents to cancel the 30 μa drawn by the more positive input terminal. Often, even though an input is unused, the user would like to know that the input is tied to a known level, and the bias network serves this purpose. The impedances of the biasing sources are sufficiently high so that the driving source and cable see virtually an open circuit when attached to the input. They are easily able to override the biasing network and apply the proper signal to the input. The biasing network is also designed to be unbalanced in such a way as to cancel the input currents mentioned above. This prevents the

need to return those currents through the ground return.

Power distribution

A second source of noise that we classify as internal, although one might disagree with this classification, is the power line noise. Consider Figure 6. If the noise appearing at the transformer inputs is unbalanced in the sense that V_{n1} and V_{n2} are equal and in phase, then a current will circulate through the loop consisting of the transformers, the primary to secondary capacitances, and the grounding between the power supplies. Assume C_1 and C_2 equal C_3 and C_4 respectively. Then, if V_{n1} and V_{n2} are equal but out of phase, the problem is less serious and a sufficiently fast regulator response with a filter network having a sufficiently low pass characteristic will correct any problems. If C_1 and C_2 do not respectively equal C_3 and C_4, then even if V_{n1} and V_{n2} are equal and out of phase, current will flow in the ground reference. Power supplies that appear to be large enough to be useful imply a power transformer having substantial

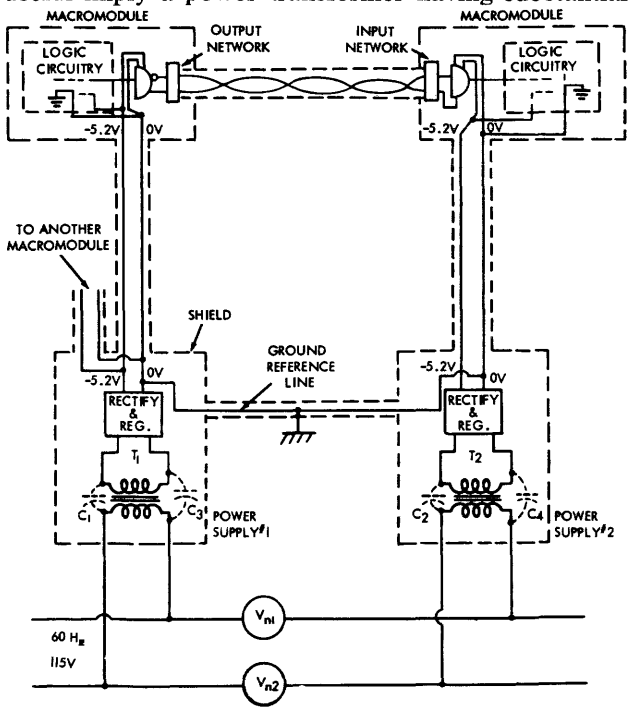


Figure 6 – A simple power supply scheme illustrating the effects of AC line noise

interwinding capacitances. In Figure 6 the reliability of signal transmission between the two logic modules at the top of the figure depends upon how much noise is present in the ground reference. In spite of the precautions of balanced signal transmission. an excess of current in the ground reference may generate a voltage drop exceeding the unbalanced noise immunity with the result that the receiving module may misinterpret the signal being transmitted. Thus, it would be desirable to keep the interwinding capacitances to a minimum and thus present the maximum impedance to power line noise generated currents.

There are several other considerations which enter into the specification of the power supply. Power consumption by the power supplies themselves can represent a substantial portion of the power dissipated in the computer. If the power supply is to be located near the modules, such as actually being attached to a section of frame, the total equipment power consumption in the vicinity of the operating modules will depend, in large part, on the supply's efficiency. The power consumption is very likely to be the factor which will limit the density of the system unless substantial liquid or forced air cooling is used. Because macromodules must be convenient to assemble, the additional complication of requiring cooling seems undesirable. In addition to efficiency, if the power supply is to be mounted in the frame, size and weight become important factors.

The preceding considerations lead to restrictions on transformer capacitance, power supply, size, weight, and efficiency. These restrictions suggest a supply in which the line voltage is rectified and filtered to give a high voltage at a low current and then converted, at a frequency which is high relative to the line frequency, to a low voltage at a high current. A block diagram of such a supply is shown in Figure 7. The conversion is achieved with the usual dc to dc converter and the efficiency is kept high by achieving line regulation of the output voltage with a pulse width regulator rather than a series regulator. The inverter transformer is small relative to a 60 cycle transformer and the capacitances assigned to it are also relatively small. Transformer droop and filter voltage drop which vary with load are compensated by a current feedback path.

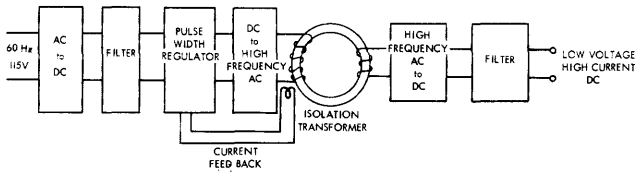


Figure 7 – Power supply block diagram

Studies on a 20 amp, 5.2 volt supply indicate that the following performance goals are practical: (1) an efficiency somewhat exceeding 70%; (2) a weight less than 10 pounds; (3) a volume of about 3½" × 4½" × 8"; and (4) less than 50 picofarads primary to secondary transformer capacitance.

A supply having these characteristics would be used in parallel with three others to power the units in a frame. An alternative approach would involve a single, larger supply designed to provide power to one frame

section. In the method of assembly in which units are not mounted in a frame, but instead interlock as described earlier, the power supply design is less restricted and has not been considered in detail.

Construction details

The macromodule itself will consist of a metal case which surrounds several printed circuit boards to which integrated circuits are soldered. The board design allows 0.3 inch^2 of board per can. A typical case size is $3'' \times 5'' \times 10''$ with a board size of $4'' \times 8''$.

The printed circuit boards consist of four layers of circuitry separated by epoxy fiberglass as shown in Figure 8. The upper two layers provide signal interconnections between the integrated circuit cans. These two layers are separated by .006 inches of epoxy fiberglass insulation, and are in turn separated from the third laver bv .032 inches of insulation. The third layer provides a ground plane which acts to confine the electric and magnetic fields of the interconnecting leads to the region between the lead and the ground plane. The coupling both between the leads themselves and between the leads and external electrical fields will be appreciably reduced by the presence of the ground plane. The fourth layer is an interdigital pattern which carries power and bias voltage to the circuit cans. It is separated by only .006 inches from the ground plane, and its capacitance to the ground plane provides some additional filtering of these d.c. voltages.

The clearance hole method of making connections between the can leads and interior layers, and between leads of the two circuitry layers has been chosen; manufacturing simplicity was the main factor in this choice.

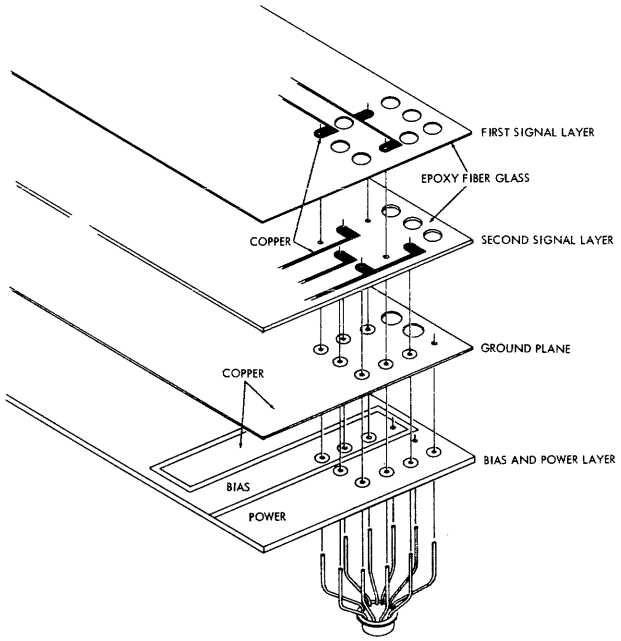


Figure 8 – An exploded view of a multilayer board

A macromodular systems simulator (MS2) *

by RICHARD A. DAMMKOEHLER
Washington University
St. Louis, Missouri

INTRODUCTION

The macromodular systems simulator is a control program and programming language implemented at Washington University to facilitate the design and subsequent realization of macromodular computer systems. The MS2 control program and language enable an engineer or programmer to describe macromodular systems and run programs on such systems simulated on a conventional digital computer (IBM 360/50). Input to the simulator consists of a standard set of function definitions for the basic macromodules, a description of the organization of the target machine, the program to be executed by the target machine, and data required by that program. Output from the present version of MS2 is a level change map (a continuous trace of the *internal* control network of the target machine as it executes its program) and the results of computations performed by the target machine. A procedure for preparing wiring tables and diagrams has been developed and will be incorporated into an improved version of the simulator.

The MS2 programming language

The MS2 programming language is, to the user, a symbolic system similar to those designed for assembly language programming with conventional machines. This correspondence was purposely maintained in order to provide a simple transition from programming to systems design. There are seven classes of MS2 commands, including the pseudo-commands which provide the capability necessary for external communication between the designer and his simulated macromodular machine.

In the following discussion the Greek symbol α will be used to designate an alphabetic or alphanumeric name assigned by the programmer or designer to a data module. The symbol β will designate a control module. Subscripts on either symbol indicate its position on an argument list.

A single data module has 12 binary positions which may be represented by either of two conventions:

(1) $\beta_{<11>}\beta_{<10>}\beta_{<9>} \ldots \beta_{<1>}\beta_{<0>}$, where the subscripts correspond to the associated power of two, or

(2) $X_{<1>}X_{<2>}X_{<3>} \ldots X_{<11>}X_{<12>}$, where the subscripts correspond to the bit ordering from left to right.

Register commands

This class of commands defines operations usually performed on a register stack or between two register stacks.

CLEAR, α	: The contents of α are set to zero by this command.
INDEX, α	: This command causes the contents of register α to be increased by 1.
COMP, α	: This command replaces the contents of register α with its one's complement.
GATE, α_1, α_2	: This command causes the contents of macromodule α_1 to be transmitted to α_2.
ADD, α_1, α_2	: This command implies that the contents of α_1 is added to α_2 and the result placed in α_2. Basic 2's complement arithmetic is used.
SHFTRZ, α	: The contents of α are shifted one bit to the right. A zero is inserted in the high order position and the low order bit is lost.
SHFTLZ, α	: Reverse of SHFTRZ.
SHFTRA, α	: The contents of α are shifted one place to the right. The low order bit of α is rotated around to the high order position.
SHFTLA, α	: Reverse of SHFTRA.
SHFTRO, α	: Same as SHFTRZ but a one

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

is inserted in the high order position.

SHFTLO, α : Reverse of SHFTRO.

SHFTRC, α : One position right shift with original high order position held constant.

SHFTLC, α : Reverse of SHFTRC.

SHFTRD, α_1, α_2 : One position right shift with high order position of α_1 replaced by the low order position of α_2, an attached data cable. α_2 is unchanged.

SHFTLD, α_1, α_2 : Reverse of SHFTRD. The low order position of α_1 is replaced by the high order position of α_2, an attached data cable.

Storage access commands

Two commands, READ and WRITE provide the facility for accessing the macromodular storage unit. Associated with each 4096 word bank of storage is a single 12-bit address register. The name of this register must appear as the second (α_2) operand of the storage access commands, as it specifies which memory bank is to be used.

READ, α_1, α_2 : This command causes the contents of the location specified by α_2 to be placed in or on data module α_1.

WRITE, α_1, α_2 : The WRITE command causes the contents of α_1 to be placed into storage at the location specified by α_2.

Screwdriver commands

Certain operations, normally performed manually, can be specified in the MS2 language. Switch settings on parameter units, and formats for the outputs of junction units are established through the use of the following commands. For example, the statement:

PARAMETER * α, 001000110001

defines a parameter unit, α, with the setting shown as the second argument. If a parameter unit is to be used as a detector setting, X's may be inserted in the second argument to indicate the 'don't care' condition. An example is shown below:

DETECTOR * α, XXXX1001XXX1

Junction units have two 12-bit inputs and one 12-bit output which may be a composite of the inputs and pre-set constants supplied by the junction unit. For example:

JUNCTION

* $\alpha_1, \alpha_2, X_{<1>}X_{<2>} \ldots 101 \ldots Y_{<1>}Y_{<2>}, \alpha_3$

defines a junction unit output α_1 with inputs from α_2 and α_3 where the $X_{<i>}$ bits are from α_2 and the $Y_{<i>}$ bits are from α_3. An alternative specification using the power of 2 convention would be

JUNCTION

* $\alpha_1, \alpha_2, X\beta_{<11>}X\beta_{<10>} \ldots 101 \ldots Y\beta_{<11>}Y\beta_{<10>}, \alpha_3$

Junction units may be activated by inserting the name of a previously defined unit as the first argument of the GATE command. For example, if T1 is the name of a junction unit output, the statement

GATE, T1, ACC

will place the current value of T1 in or on data module ACC.

Conditional commands

Modules with settings or parameters defined previously may be interrogated by the following MS2 statements:

DETECT, $\alpha_1, \alpha_2, \beta_1, \beta_2$

where α_1 = name of the module being detected
α_2 = name of previously defined detector
β_1 = control path for TRUE response
β_2 = control path for FALSE response

A decoder may be interrogated by a statement such as

DECODE, $\alpha_1, \alpha_2, \beta_0, \beta_1, \beta_2, \beta_3 \ldots \beta_7$

where α_1 = bits to be decoded written in the form $\beta_{<11>}\beta_{<10>}\beta_{<9>}$ or $X_{<1>}X_{<2>}X_{<3>}$
α_2 = name of the data module
(note: this may not be a multiple length module)
β_0 = control path to be taken if the bit pattern $(\alpha_1) = 000$
β_1 = control path to be taken if the bit pattern $(\alpha_1) = 001$

•
•
•

An overflow condition on an adder may be detected by the statement

ADDOV, $\alpha_1, \beta_1, \beta_2$

where α_1 is the name of the register stack containing the adder, β_1 is the TRUE (OVERFLOW) return and β_2, the FALSE (NO OVERFLOW) return. Use of the ADDOV command does not reset the overflow indicator.

The contents of two registers may be compared by attaching the data output cable of one register to the data terminal of a detector positioned on a second register. The following MS2 command is used for this operation:

COMPARE, α_1, α_2, β_1, β_2

where α_1 = name of register on which the detector is positioned
α_2 = name of the register providing detector input
β_1 = TRUE control branch
β_2 = FALSE control branch

Logic function commands

This class of commands specify operations performed in a register stack by the macromodular function unit.

BITSET, α_1, α_2 : The bit pattern of register α_2 is modified by the pattern of α_1 using OR logic. Result in α_2.

BITCLR, α_1, α_2 : If the input bit from α_1 is zero, the corresponding bit in α_2 is unchanged. If the input bit from α_1 is one, the corresponding bit in α_2 is set to zero.

BITCOMP, α_1, α_2 : If the input bit from α_1 is zero, the corresponding bit in α_2 is unchanged. If the input bit from α_1 is one, the corresponding bit in α_2 is complemented.

AND, α_1, α_2 : Perform an AND operation between α_1 and α_2 bit patterns.

Control commands

Additional commands for the control macromodules are

MERGE, β_1, : which implies the merging of control at point β_1, and

CALL, β_1, β_2, : where β_1 is the point at at which a reentrant control string begins, and β_2 is the point to which control will be passed following the execution of that control string.

If a reentrant control string contains more than one exit point, (e.g., at detector or other decision module) the decision call macromodule should be used. The simulator command format is

DCALL, β_1, β_2, β_3, : where as before β_1 specifies the beginning of a control string, and β_2 and β_3 specify the true and false branch points.

A control string may be split into two independent strings by the use of the simulator command

SPLIT, β_1, β_2, : which specifies that two functions are to be initiated simultaneously; one at a module labeled β_1 and the second at β_2.

Any two control strings can be re-joined using the macromodular rendezvous unit simulated by a command of the following form

WAIT, β_1, β_2, : where β_1 is the name of a particular rendezvous unit and β_2 is the point to which control will pass after two control strings have been rejoined.

Pseudo-commands

Six pseudo-commands have been implemented in order to facilitate the use of MS2. They are:

INPUT, α : The INPUT command causes the next octal value, delimited by an apostrophe, to be read from the input stream and placed in data module α.

OUTPUT, α : The OUTPUT command prints the contents of data module α in octal format.

RETRN, α : The RETRN command passes control from the target machine to the MS2 control program. The argument α specifies a data module (usually the instruction counter), the contents of which will be printed at the time the return is executed.

TRUE : The TRUE command has no arguments as it is used as an argument for the DCALL command when a decision call unit is imbedded in a re-entrant control string called by other target machine strings.

FALSE : A command used to identify the FALSE return from a decision

call unit imbedded in a re-entrant control string.

LOCK, β_1, β_2, : The LOCK command is a pseudo command which marks the position of a priority selector. The selector module interlocks a control beginning at point β_1 and the second argument β_2 specifies the point to which control passes after the β_1 string has been executed and the interlock released.

Multiple length specification

The user may define and use multiple length data modules by including a slash (/) in argument strings of the Register, Memory Access, Conditional, and Logic Function commands. For example, the MS2 statement:

GATE,A1/A2/A3,BX/BY/BZ

specifies the gating of a 36-bit data module, with twelve bit sections A1, A2, and A3, to another 36-bit data module, consisting of sections BX, BY, and BZ. Up to 9 twelve bit sections may be coupled with this convention.

The user may 'assemble' multiple length detectors, junction units and parameter units defined by

PARAMETER*A1,100011001110
PARAMETER*A2,100000001110

to create a 24-bit parameter unit consisting of two 12-bit sections A1 and A2. These sections may be used independently or as a single 24-bit unit as shown in the following example:

TEST* COMPARE, A1,M, NEXT, NOT
NEXT* COMPARE, A1/A2, ACC1/ACC2, GO, NOGO
•
•
•

The MS2 control program

Implementation

The simulator control program is implemented in TRAC,[1] a string processing language, selected because of the similarity between the macromodular control network and Mooers' concept of an active string. But using a combination of TRAC primitives, it is possible to exactly duplicate the functional characteristics of most of the present set of macromodules. The exceptions are the control branch, a macromodule used to initiate parallel control networks, and the interlock module which functions as a priority selector at the intersection of multiple control paths. As parallel operations must be simulated sequentially on a serial machine, the functions of these two modules are handled internally by the simulator control program. However, in order to preserve the validity of the level change map and wiring tables, both control branch and interlock macromodules can be included in the description of the target machine.

Statements describing the internal organization of a computer to be simulated by MS2 are prepared in free-form format using an asterisk (*) as a delimiter for labels of unique control points (control paths) and the character ' (an apostrophe) to mark the end of each statement. They are read by the control program, analyzed syntactically, and converted to active TRAC strings prior to execution. Unlabelled statements are concatenated with the preceding labelled statement producing a sequential string corresponding to a single control path in the target machine. A complete machine definition consists of one or more such strings, existing as separate within primary core. Linkages between strings are completed at execution time by the control commands described in Section II-F.

TRAC definitions of the basic macromodular functions are also stored internally as active strings as shown in Table I. A complete listing is given in Technical Report No. 18.[2] During execution, an active target machine string initiates requests for the macromodular functions in the sequence indicated by the MS2 machine description. Arguments contained in the target machine string replace the special characters $\rightarrow_i$ marking the position of the *ith* argument, and the resulting string is evaluated by the control program. An example of the 'evolution' of an active MS2 string from input through execution is shown in Table II.

The current version of the control program was implemented in BAL, the *b*asic *a*ssembly *l*anguage for the IBM 360 series, and is run under a modified R.A.C.S. (*R*emote *A*cess *C*omputing *S*ystem) which provides for conversational interaction between the 360/50 and remote consoles. During MS2 runs, the remote console is used as the operator's console for the target machine, or alternatively as a reader/printer/punch station.

Table I. Some simple TRAC definitions of macromodular functions

#(DS,*CLEAR*, #(DS, $\rightarrow_1$,0000)))
#(DS,*GATE*,(#(DS,$\rightarrow_2$,##(CL,$\rightarrow_1$,)))

#(DS,*INDEX*,(#(DS, →1,#(AD,##(CL, →1, 0001))))
#(DS,*ADD*,(#(DS,→2,#(AD,##(CL,→1,)## (CL, →2)))))
#(DS,*MERGE*,(#(CL, →1)))
#(DS,*CALL*,(#(CL, →1(#(CL, →2)))
#(DS,*COMPARE*,(#(CL,#(EQ,(#(CL, →1)), (#(CL,→2)),→3,→4))))
#(DS,*INPUT*,(#(DS, →1,##(RS))))
#(DS,*OUTPT*,(#(PS,----##(CL, →1))))

Table II. Evolution of an active MS2 string

Structural form	Description
#(DS,GATE,(#(DS, →2,## (CL, →1))))	TRAC definition of the gate macromodule
START*GATE,P,S'	Input form of MS2 target machine description
#(DS,START,(#(CL, GATE,P,S)))	Internal form of MS2 statement
#(CL,GATE,P,S)	Result of initialization of target machine by #(CL,START)
#(DS,S,##(CL,P))	Result of #(CL, GATE,P,S)

At the beginning of a simulation run approximately 180K bytes of storage are available for the target machine description. The remaining 76K bytes are used by the operating system (~72K) and the control program. The augmented TRAC system, an integral part of the control program, occupies less than twenty-five hundred bytes of core. Storage of target machine data, programs and results is managed by the control program in order to conserve core storage in the 360/50. Such information is stored in octal representation with 8 octal digits occupying 1 word (4 bytes) of 2 μsec core. Packing and unpacking are performed by two new primitives, rm and wm (*r*ead *m*emory and *w*rite *m*emory), added to the TRAC function set.

Control program commands

The MS2 control program commands are defined in terms of the TRAC primitives, and in some cases, for convenience, use the macromodular function definitions as well. Present control program commands are:

INITIAL * α — This command calls the initialization routine which edits and analyzes all MS2 statements. α is an arbitrary identification parameter.

DEFINEBLOCK * $\beta_0,\beta_1,\ldots\beta_{31}$ — This command defines a block β_0 consisting of strings β_1 through β_n. $n \leq 31$

STOREBLOCK * β — This command stores a block β in disk storage.

FETCHBLOCK*β — This command retrieves a block of strings β from the disk and re-establishes the definitions of its constituent strings β_1 through β_n.

MAP* α — This command activates the mechanism for tracing the internal control of a target machine. α is an arbitrary identifier.

NOMAP* α — Use of this command deactivates the internal control mapping mechanism.

$\beta * \alpha$ — A command of this form passes control to any previously defined string β.

DISCUSSION

The simulator has been used to evaluate a number of macromodular realizations of specialized computer designs including a Valgol II machine[3] and the compiler-compiler machine[4] presented in the following paper. Its principal advantages are related to the ease with which a programmer may describe the organization of his target machine, and the flexibility of implementation which allows additional macromodular functions to be specified as active TRAC strings within the existing MS2 environment. This latter capability has already been of proven value, in that it provided a mechanism for maintaining a correspondence between the evolving macromodular technology and concurrent efforts to develop software for target machines.

Limited use of MS2 has been made in conjunction with computer design and programming courses offered by the department of Electrical Engineering and the department of Applied Mathematics and Computer Science. In these instances, MS2 descriptions of commercially available machines were developed by the students, and used to obtain solutions for problems assigned in class. Although this ability to verify the results of programs written for a variety of conventional machines has in practice been a valuable supplement to normal laboratory experience, it also demonstrates (indirectly) the design flexibility inherent in the macromodular concept, and the functional completeness of the existing set of processor modules.

REFERENCES

1 C N MOOERS L P DEUTSCH
TRAC, a text handling language
Association for Computing Machinery Proceedings of the 20th National Conference 229-246 1965

2 R A DAMMKOEHLER R A COOK A R RAWIZZA
TRAC implementation of macromodular functions
Technical Memorandum No 18 Computer Research Laboratory Washington University St Louis Missouri January 15 1967

3 A R RAWIZZA
The valgol II machine
Masters Thesis Sever Institute of Technology Washington University St Louis Missouri 1967

4 W E BALL
A macromodular meta-II machine
Proceedings of the Spring Joint Computer Conference 1967

A macromodular meta machine*

by WILLIAM E. BALL
Washington University
St. Louis, Missouri

INTRODUCTION

The macromodular concept[1] makes it possible to easily change the hardware design of a computer. However, a change in computer design usually implies that any existing software will no longer function correctly. It appears, then, that a flexibility in the software, equivalent to that now available in the hardware, must somehow be obtained. One possible partial solution to this problem is to separate some of the software activities, such as the compiling function, from the main computational machine. This suggests that what is needed is to add a "compiler module" to the set of available macromodules. That is, a small fixed macromodular computer is needed which would accept as input a source language specification and a target machine specification. It would then input a source language program for compilation. Its final output would be a machine language program for direct execution in the target machine. This paper describes in detail the results of an effort to design such a computer, called the "meta machine." It also illustrates the relative ease with which a new computer may be designed when the macromodular system is used.

The basic algorithm by which the meta machine is to operate must be both simple and flexible. The simplicity is required for the reduction of both expense and complexity. The flexibility is required for easy adaptation to source languages and target machines. One approach that seems to fit these requirements is described by Schorre.[2] His syntax oriented compiler writing language, along with specifications for the Meta II machine, represents the type of system proposed for the compile module.

The next section describes the extensions and modifications made to Schorre's Meta II language to adapt it to the present requirements. Following this syntax specification section, the meta machine organization is described. Finally, the implementation of the meta machine organization by macromodules is presented.

Syntax specification

Table 1 lists the complete basic compiler for the meta machine, written in its own input language for execution on the meta machine. The normal steps for the use of the overall system are illustrated in Figure 1. The basic compiler is used to generate a special purpose compiler in step 1. In step 2 the special purpose compiler is used to generate the final object code for the target computer.

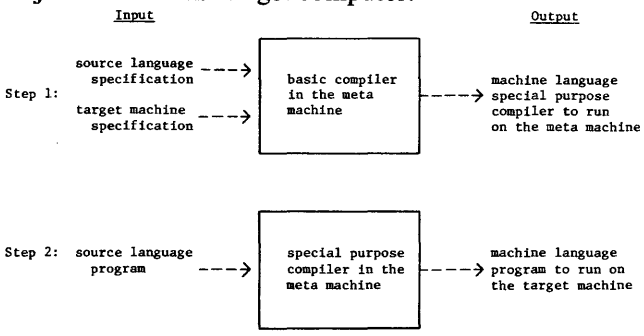


Figure 1 – Overall use of the meta machine

Note that the source language specification includes both syntax and semantic rules. Any changes in the target machine can now be readily handled by changing the input specificiations to the basic compiler and then regenerating the appropriate special purpose compiler. The input language for the basic compiler is essentially the Meta II language proposed by Schorre.[2]

This system represents a top down, no back up, type of syntax analysis algorithm. However, some of the ideas described by Oppenheim and Haggery,[3] as incorporated in their Meta 5 language, have also been used. One significant difference is that the basic compiler of the meta machine is to generate machine language code directly, not assembly language code.

The source language specification consists of two parts:

1. Symbolic target machine instructions and their bit code equivalent.

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

```
.DEFINE
BE =1000;
NO =1100;
GN1=1110;
GN2=1120;
OUT=1130;
DTV=1140;
STA=1150;
ATI=1160;
SAV=1170;
SET=1200;
RET=1210;
END=1220;
BOT=1230;
CCD=1240;
CCS=1250;
CDS=1260;
BLK=1270;
CLS=2000;
TC  =3000;
CL  =4000;
B   =5000;
BT  =6000;
BF  =7000;
.SYNTAX PROGRAM ;
PROGRAM='.DEFINE'$DEF'.SYNTAX'ID.OUT(*,'CLL'/'BE'/)';'.OUT('END'/)$ST'.END';
DEF = ID '=' OCTNUM ';' .SAVE ;
OCTNUM = $(OCTDIG .OUT(*5)) ;
ST = ID .LABEL * '=' EX1 ';' .OUT('RET'/) ;
EX1 = EX2 $('|' .OUT(*1,'BT'/) EX2) .LABEL *1 ;
EX2=(EX3.OUT(*1,'BF'/)|OUTPUT)$((EX3.OUT('BE'/)|OUTPUT)).LABEL*1;
EX3=ID.OUT(*,'CLL'/)|STRIN.OUT('BLK'/)|'.QUOTE'.OUT('QOT'/)|'.EMPTY'
        .OUT('SET'/)|'.BLANK'.OUT('BLK'/)|'.SAVE'.OUT('SAV'/)
        | '(' EX1 ')' | '$' .LABEL *1 EX3 .OUT(*1,'BT'/'SET'/) ;
OUTPUT = '.OUT' '(' $(OUT1 (',' | '/' .OUT('OUT'/)) ')' | '.LABEL' OUT2;
OUT1 = '*1' .OUT('GN1'/) | '*2' .OUT('GN2'/) | '*3' .OUT('CCD'/) |
        '*4'.OUT('CCS'/)|'*5'.OUT('CDS'/)|'*'.OUT('ATI'/)|
        STROUT .OUT('DTV'/);
OUT2 = '*1' .OUT('GN1'/'STA'/)  | '*2' .OUT('GN2'/'STA'/) |
        '*' .OUT('STA'/);
ID = LETTER .OUT(*4) $( (LETTER | DIGIT) .OUT(*4) ) ;
STRIN = .QUOTE .OUT('SET'/) $(CHAR .OUT(*3,'TC'/)) .QUOTE ;
STROUT = .QUOTE$(CHAR.OUT(*3,'CLS'/)).QUOTE;
CHAR = LETTER | DIGIT | SPCHAR ;
LETTER = 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|
        'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z' ;
DIGIT = OCTDIG | '8' | '9' ;
OCTDIG = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ;
SPCHAR = '-' | '(' | '|' | '$' | '*' | ')' | ',' | '=' | ' ' ;
.END
```

TABLE I

2. Syntax rules for the source language and their corresponding semantic rules for the target machine symbolic instructions.

The first input part has the form:

.DEFINE

<symbolic instruction> = <octal number>;

This last line is repeated until all target machine instructions have been defined. The second input part has the form:

.SYNTAX <identifier of language head>; <series of statements defining the syntax and semantic rules>

.END

The statements actually define recursive subroutines. The identifier on the left side of the '=' defines the name of the corresponding subroutine. The use of an identifier on the right side of the '=' causes a call to that subroutine to be generated. The result of the subroutine is communicated back to the calling program by means of a true/false switch. A true return indicates that the syntactic unit described by the subroutine has been found in the input. A false return indicates that the syntactic unit has not been found. A literal string appearing in a syntax rule implies that the same string must appear in the input for the particular rule to apply.

The semantic rules are described in two ways. First, by the expression .LABEL, to indicate that the next identifier is a label whose location is now defined. Second, by the expression .OUT, to generate target machine code. The function .OUT may have any number of arguments separated by either a ',' or a '/'. The former causes the bit pattern of the preceding argument to be added into the output register. The latter also adds the argument bit pattern to the output register, but in addition causes the contents of the output register to be written out. As an example,

.OUT (*1,'BT'/'SET'/) will cause the target machine bit patterns for two instructions to be put out:

BT *1
SET

A number of additional codes are available in the basic compiler to make the necessary character and symbol manipulation possible:

* Use the contents of the current symbol stack.
*1 If a label of type 1 has been generated on this recursion level, use it. Otherwise generate a new label of type 1. Put the label into both the current symbol stack and the output register.
*2 Same as *1, only for a label of type 2.
*3 Add the last recognized character to the output cell.
*4 Add the last recognized character to the current symbol stack.
*5 Shift and add the last recognized octal digit to the value part of the current symbol stack.
$ Repeat the following syntactic unit 0 or more times.
.BLANK Delete blanks from the input string.
.SAVE Save the current symbol stack in the definition table.
.QUOTE Test the input string for a string quote.
.EMPTY A syntactic unit that is always true.

The technique of label generation, the current symbol stack, the definition table, and the output register are explained in the next section.

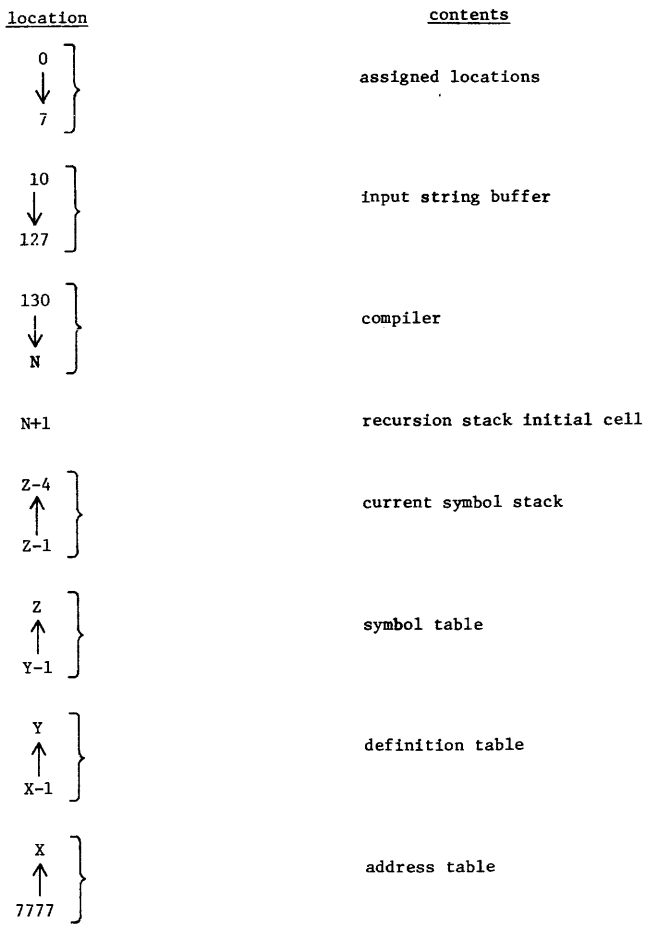


Figure 2 – Meta machine memory allocation

Meta machine organization

The memory module used by the meta machine contains 4096 words, each 12 bits in length. Figure 2 illustrates how the memory is allocated during an actual compilation run. All memory addresses in that figure and in the following discussion are expressed as octal numbers. Each word in the memory module normally contains either one machine instruction, one character, or one absolute machine address. Certain exceptions to this, however, will be noted later.

Starting from the low end of memory, the first eight locations have definite assignments. As the compilation proceeds, the memory location currently being filled in the target program is recorded in location 0. This location is initially set to what the first location address of the target program is to be. For the meta machine, all programs start at location 130.

Locations 1 and 2 record the boundaries of the running program. For later reference, the last location used by the program in lower core is called "N." This is the value recorded in location 1. The program itself will occupy all of the space from 0 to N. A number of the machine instructions, in particular the transfer instructions, require an associated address table. The address table for the running program is stored in the upper part of memory starting in the last location, 7777. The address table then extends downward to some location which will be called "X." Location 2 records the value of X. These two locations are the only values that need to be set by whatever system is used to initially load the program.

During the execution of the program two variable lengths of storage are required. The first, called the recursion stack, keeps track of subroutine entries and exits, and any generated labels. The recursion stack begins at the first free cell of lower memory (N+1), and extends upward as far as needed. The exact use of the stack is described under the instructions CLL and RET in Table 2.

Table 2 – Meta machine instruction set

instruction	*octal code*	*operation*
BE	1000	Branch to the error routine if the switch register is set false.
NOP	1100	Continue on to the next instruction.
GN1	1110	Test to see if a label of type 1 is available at the current recursion level. If not, a new label is generated and saved in the recursion stack. The available label is then looked up in the symbol table. The target address table index of the label is then added to the output cell.
GN2	1120	Exactly the same as GN1, only for a label of type 2.
OUT	1130	The target program instruction counter and the contents of the output cell are written out. The target machine address is incremented and the output cell is cleared.
DTV	1140	The contents of the current symbol stack are looked up in the definition table. The value of the symbol is then added to the output cell.
STA	1150	The contents of the current symbol stack are looked up in the symbol table. The address of symbol is then set to the value of the target machine instruction counter.
ATI	1160	The contents of the current symbol stack are looked up in the symbol table. The target address table index of the symbol is then added to the output cell.
SAV	1170	Decrement by four the definition table and symbol table pointers (Y and Z).
SET	1200	Set the switch register true. Save the current input string pointer in the past input string pointer cell.
RET	1210	Return from a recursive subroutine. This pops three cells off of the recursion stack. The first two were cells to hold the generated labels of type 1 and 2. The third cell contains the actual machine address of the corresponding call instruction.
END	1220	Write out the target machine address table and then terminate the program.
QOT	1230	The input string is tested for the occurence of a string quote. The switch register is set true if it is found, and the input string pointer is advanced. The switch register is set false if a string quote is not found.
CCO	1240	The current input character bit pattern is added to the output register.
CCS	1250	The low order six bits of the input string character are copied into the next available character position of the current symbol stack, up to a maximum of five characters. Characters beyond the fifth are ignored.
CDS	1260	The input string octal digit is copied into the value part of the current symbol stack in the symbol table.
BLK	1270	The input string is tested for the occurence of a blank. If one is found, the input string pointer is advanced and the test repeated.
CLS	2xxx	The low order six bits of the literal character represented by the low order bits in the instruction are copied into the next available character position of the current symbol stack, up to a maximum of five characters. Additional entries beyond five characters are ignored.
TC	3xxx	If the switch register is false, continue on to the next instruction. If the switch register is true, test the input string for the occurence of the literal character represented by the low order bits in the instruction. If it is found, advance the input string pointer. If it is not found, set the switch register false and restore the current input string pointer to the value saved in the past input string pointer cell.
CLL	4yyy	A recursive call of a subroutine. Three cells are pushed into the recursion stack. The first contains the current instruction counter. The next two contain zeros to allow for any label generation in the called routine. Control is then transferred to the location given by the address table index yyy.
B	5yyy	Control is transferred to the location given by the address table index yyy.
BT	6yyy	If the switch register is true, transfer control to the location given by the address table index yyy. If the switch register is false, continue on to the next instruction.
BF	7yyy	If the switch register is true, continue on to the next instruction. If the switch register is false, transfer control to the location given by the address table index yyy.

The second variable block of storage is required to save symbols and identifiers as they are found by the compiler. They symbol and identifier storage area is divided into two parts. The first part, called the "definition table," starts immediately below (X-1) the end of the address table. It is the purpose of the definition table to give the target machine bit pattern equivalent to any symbolic instruction that is generated by the compiler. Location 3 records the last location, called "Y," used by the definition table. Thus, the definition table extends from location X-1 downward to location Y, where the extent is fixed by the number of defined symbolic instructions in the source language. By the requirements of the

compiler program, all defined symbols must come before the start of the actual syntax specifications of the source language, hence all defined symbols will be stored in the definition table before the first appearance of an identifier.

As the syntax rules are processed by the running program, identifiers will be found or labels will be generated. A symbol table is used, starting immediately below the definition table, to record these identifiers as they are found. The symbol table extends from location Y-1 downware to location "Z." Location 4 in lower memory records Z, the last location used by the symbol table.

Locations 5, 6, and 7 serve to hold values that are needed by the running program. Location 5 saves a previous value of the input string pointer. If the program checks for a given string in the input, and fails to find it, the input string pointer can be backed up by the use of this recorded value. Locations 6 and 7 are simply counters. They count the number of generated labels, of two distinct types, so that a new unique label may be provided whenever one is required.

The input string being processed by the running program is stored in locations 10 to 127. These locations actually represent one word for each column of one punched card. The input string pointer is initially set to 10, the address corresponding to the first column of an input card. As the input string pointer is incremented, it scans the card until it reaches location 127 (card column 80). The next time the machine detects that the input string pointer is to be incremented, a new card is read instead. The input string pointer is then reset to location 10. This system requires that any quoted string in the input occur completely on a single card. Otherwise the input string back up mechanism would operate incorrectly when the input string fails to match the expected value. Location 130 contains the first instruction of the running program. All of the currently implemented instructions in the meta machine are defined in detail in Table 2. There are 23 instructions, grouped as follows:

Control type instructions:
- B - branch
- BE - branch to error routine if false
- BF - branch if false
- BT - branch if true
- CLL - call subroutine
- END - end of program
- RET - return from subroutine
- SET - set switch true

Character and stack manipulation instructions:
- ATI - find address table index
- BLK - delete blanks
- CCO - copy character into output
- CCS - copy character into stack
- CDS - copy digit into stack
- CLS - copy literal into stack
- DTV - find definition table value
- QOT - test for quote
- SAV - save current symbol
- STA - set symbol table address
- TC - test character

Miscellaneous instructions:
- GN1 - generate type 1 label
- GN2 - generate type 2 label
- NOP - no operation
- OUT - output

Table 3 illustrates the general nature of the entries that occur at the high end of memory, in both the definition table and symbol table. The entries always occur in groups of four words. The word with the highest memory address, the first word of the entry, contains two six bit fields. The first field contains a count of the number of characters (letters or digits only) that the identifier contains, up to a maximum of seven. The second field contains the low order six bits of the eight bit pattern of the first character in the identifier. Words two and three of the entry contain, respectively, the code for characters two and three and characters four and five. Symbols of five characters or less thus have a unique entry. Identifiers that are longer than five characters, and match through the first five, must have different lengths to have unique entries in the table.

The fourth word of the table entries may have any one of three different values. For a definition table entry, the fourth word will contain the tartet machine bit pattern corresponding to the instruction represented by the symbolic entry. For a symbol table entry, the fourth word will contain the target machine ifier that has appeared only in the form of a call or transfer operand. Once an identifier appears as a label, the fourth word is filled in with the value of the target machine instruction counter. This will eventually be the actual address table entry when the target program is executed. References to this address table entry are made by using its address table index. This index is defined to be its memory location in the address table minus 7000. If the address table index of any symbol is required for the completion of an instruction, it is found by:

1. Locate the desired symbol in the symbol table. Assume the fourth word of the entry is found at location L.

2. Compute the address table index by:

$$1000 - \frac{(Y-L)}{4}$$

Table 3 – Symbol table entries

location	*contents*
L	address or value of symbol
L+1	(character 5) (character 4)
L+2	(character 3) (character 2)
L+3	(character 1) (character count)

The entries in the definition and symbol tables are actually created in the current symbol stack. This stack always occupies the next four locations below the symbol table. When it is desired to locate a given symbol in the table, the symbol is created, as a standard table entry, in the current symbol stack. This entry in the current symbol stack is then compared, entry by entry, with all of the entries in the table. If a match is found, a pointer is set to point to the fourth word of the appropriate entry. If a match is not found, the end of table pointer is decremented by four, including what was the current symbol stack in the table. Thus, the entry that was being searched for is now found. Note that this also automatically displaces the current symbol stack downward in the memory by four locations.

The next section describes how the meta machine was constructed by interconnecting standard macromodule units.

Meta machine implementation

The design of the meta machine was developed by programming the interconnections of the macromodule units using the MS2 simulation language.[4] By using this technique the machine could be tested, modified, and checked out, all by simulation, before any macromodule hardware had to be used. However, a knowledge of the hardware is necessary before the simulation can be discussed.[1]

Besides the memory module mentioned in the last section, one of the most significant units is the register module. The basic module holds 12 bits. It has provisions for clearing, indexing, and complementing its contents. It is also possible to add other important actions. For example, adding a data gate module allows a word to be entered into the register. Adding a detector module allows a test for a particular bit pattern to be made on the contents of the register. In all of these units, however, there is one central concept, the control path, that governs the action of the module. There are normally two connections made to a macromodule. One connection, for the control path, carries the information that tells the unit when it should perform its action. The other connection, for the data path, carries the information necessary for the module to operate.

Table 4 contains a list of all of the register modules used in the meta machine, with both their function and reference name. In designing a computer such as the meta machine one important decision concerns the number of registers to use. Quantities that are referenced frequently may be retained in registers or they may be retrieved from fixed memory positions when needed. The former choice offers faster operation, neater logic, and far more expense. The latter choice reduces the number of expensive macromodular units required (such as registers), while increasing the number of cheaper units (data gates and branches). The approach taken with the meta machine falls roughly midway between the two extremes. A generous supply of registers is used to make the wiring for most of the instructions reasonably simple. However, a number of necessary values are still retrieved from memory on demand. Thus, this particular design represents a compromise between an expensive, fast machine and a cheap, slow machine.

Table 4 – Register Assignments

Register Label	*Function*
RRA	memory address register
RRM	memory I/O register
RR1	instruction counter
RR2	switch register (0≡true, ≠ 0≡false)
RR3	output register
RR4	recursive stack pointer
RR5	input string pointer
RR6	utility
RR7	utility
RR8	utility
RR9	utility

Another type of macromodule that is extremely useful is the call unit. This module makes it possible for more than one control path to activate one given control path, and yet retain its own identity. Thus it is possible to consider "hardware subroutines" that may be called by other parts of the computer at any time. For the cost of one call unit it is possible to avoid duplicating large blocks of modules. An extension of the call unit concept that includes the possibility of a branch in the control path is the decision call unit. This module allows two possible returns, as from a detector unit, while still maintaining the identity of the original activating control path. Figures 4 and 5, to be discussed in detail later, contain some

schematic diagrams of registers, call units, control paths, and data paths. The conventions used in the diagrams are illustrated in Figure 3.

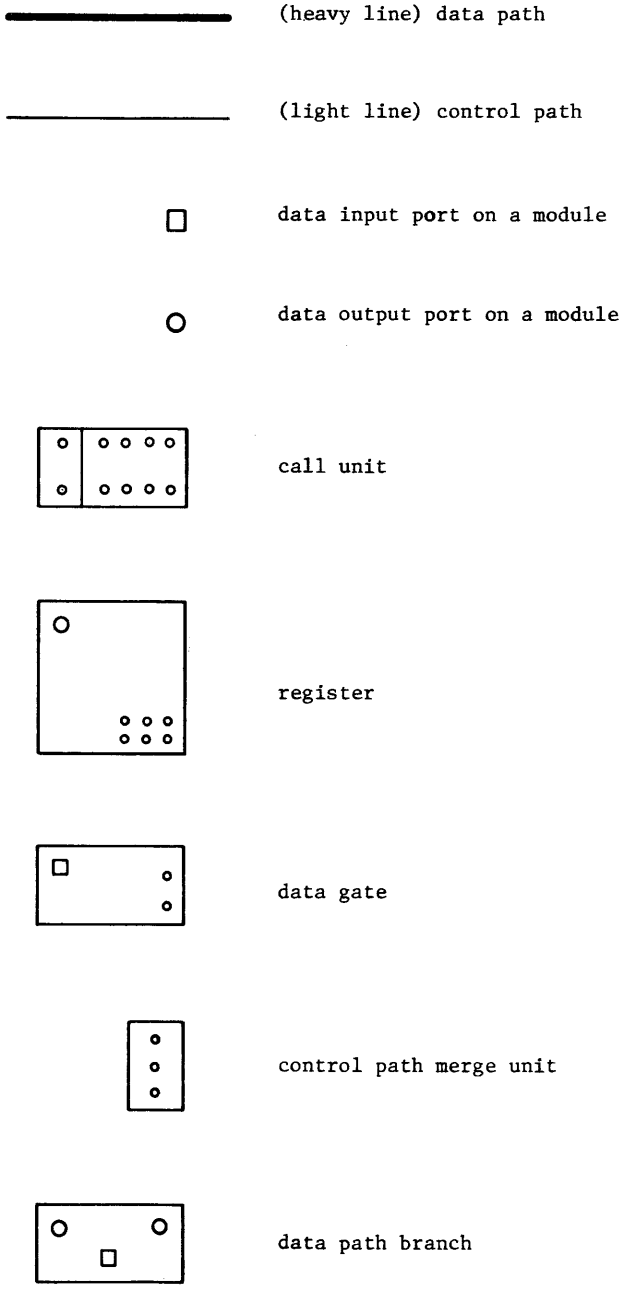


Figure 3 – Macromodular diagram conventions

Labels on the various units and control paths are identical to the corresponding names used in the MS2 listing in Table 5.

The MS2 simulator represents the possible connections between modules in different ways. Data path connections, from register to data gates, are written explicitly. For example, to gate the contents of register RR5 to register RRA requires the statement:

GATE, RR5, RRA'

This action will be performed when the control path reaches this statement. Normally this would be after the previous statement was activated. If the control path is not sequential, then the control path itself must be given a unique label. Thus if the control path labeled BLK calls a subroutine with initial control path labeled R5RD, and the control path labeled B1 is to be active after the return, the simulation statements would be:

```
BLK*   CALL, R5RD, B1'
B1*    DETECT, RRM, DD5, B2, INTP1'
R5RD*  GATE, RR5, RRA'
       READ, RRM, RRA'
```

These statements describe part of the connection diagram in Figure 5.

A listing of the input to the MS2 simulator is presented in Table 5. This listing completely defines the meta machine since it specifies all of the data and control paths between the macromodule units. The initial active control path is labeled BEGIN. The main loop begins at INTP1', where the instruction counter is incremented by 1. The next instruction is then obtained from memory and decoded. Decoding causes a many-way branch in the control path, each branch leading to the execution of one of the machine instructions. Most of the instructions then terminate by merging the control path back to INTP1. Figure 4 illustrates how this procedure is implemented using the actual macromodule units. The following description of Figure 4, however, may be easily followed in the simulation listing of Table 5. The initial control path, labeled INTP1, merges with another control path, labeled NIP, to allow for a no operator loop. The path then indexes the contents of the instruction counter register, RR1. Some of the instructions do not require indexing this instruction counter, hence the path now merges with an alternate entry path, INTPO. This combined control path activates, by means of a call unit, a hardware subroutine labeled R1RD. This subroutine gates the contents of the instruction counter into the memory address register, and then calls the subroutine MTORM. This subroutine reads the contents of the memory location contained in RRA, and then gates it into register RRM. On return from the subroutine R1RD, the control path 114 will be active. This path causes the decoding units to look at the instruction now contained in the register RRM. The first decoding unit looks at the initial three bits (numbered from left to right 1, 2, 3) of the instruction. A value of 0 or 2 through 7 causes the corresponding instruction control path to be activated. A value of 1, however, requires the second decoder to look at bits 4, 5, 6. Only three valid values may be found: 0 identifies the in-

Table 5

The Meta Machine Expressed In The MS2 Language

```
DETECTOR*      DD1,    000000000000'
DETECTOR*      DD2,    000001111101'
DETECTOR*      DD3,    000000000000'
DETECTOR*      DD4,    000000000000'
DETECTOR*      DD5,    000001000000'
DETECTOR*      DD6,    000000000000'
JUNCTION*JJ1,RRM,      111X<4>X<5>X<6>X<7>X<8>X<9>X<10>X<11>X<12>,NULL'
JUNCTION*JJ2,RRM,      Y<7>Y<8>Y<9>Y<10>Y<11>Y<12>X<7>X<8>X<9>X<10>X<11>X<12>,RR
JUNCTION*JJ3,RRM,      X<4>X<5>X<6>X<7>X<8>X<9>X<10>X<11>X<12>Y<10>Y<11>Y<12>,RR
JUNCTION*JJ4,RRM,      Y<1>Y<2>Y<3>X<4>X<5>X<6>X<7>X<8>X<9>X<10>X<11>X<12>,RR3'
JUNCTION*JJ5,RRM,      X<1>X<2>X<3>X<4>X<5>X<6>Y<7>Y<8>Y<9>Y<10>Y<11>Y<12>,RR5'
JUNCTION*JJ6,RRM,      000X<4>X<5>X<6>X<7>X<8>X<9>X<10>X<11>X<12>,NULL'
JUNCTION*JJ7,RR7,      000000X<1>X<2>X<3>X<4>X<5>X<6>,NULL'
PARAMETER*     PP0,    000000000000'
PARAMETER*     PP1,    000000000001'
PARAMETER*     PP2,    000000000010'
PARAMETER*     PP3,    000000000011'
PARAMETER*     PP4,    000000000100'
PARAMETER*     PP5,    000000000101'
PARAMETER*     PP6,    000000000110'
PARAMETER*     PP7,    000000000111'
PARAMETER*    PP10,    000000001000'
PARAMETER*  PP130,      000001011000'
ATI*        CALL,      LOOK,      A5'
  A5*       CALL,      COMP6,     A6'
  A6*       CALL,      P3RD,      A7'
  A7*       ADD,       RR6,       RRM'
            COMP,      RRM'
            CALL,      SHFTM,     A9'
  A9*       CALL,      SHFTM,     A10'
  A10*      BITSET,    JJ6,       RR3'
            MERGE,     INTP1'
  A1*       CALL,      R6RD,      A2'
  A2*       CALL,      SETR3,     INTP1'
  A3*       CALL,      PORD,      A4'
  A4*       CALL,      WTR6,      INTP1'
B*          GATE,      JJ1,       RR1'
            CALL,      R1RD,      B17'
  B17*      GATE,      RRM,       RR1'
            MERGE,     INTP0'
```

struction BE; 1 and 2 indicate further decoding, by inspecting bits 7, 8, 9 to be required. Thus the third and fourth decoding units complete the identification of the instruction and activate the corresponding control paths for execution.

While it is a relatively straight forward task to transform the complete simulator listing into an actual wiring diagram, the result tends to become too complex to read. Figure 5 presents a portion of the detailed wiring diagram illustrating the implementation of three instructions. Consider first BLK, the instruction that deletes blanks from the input string. The entry point from the decoders in Figure 4 is to the control line labled BLK. The first action is to merge with a return path to allow for multiple blanks. Then a call module is used to obtain the current input string character. Register 5 contains the address of this character, so the hardware subroutine R5RD performs a function quite similar to the subroutine R1RD previously described. The contents of RR5

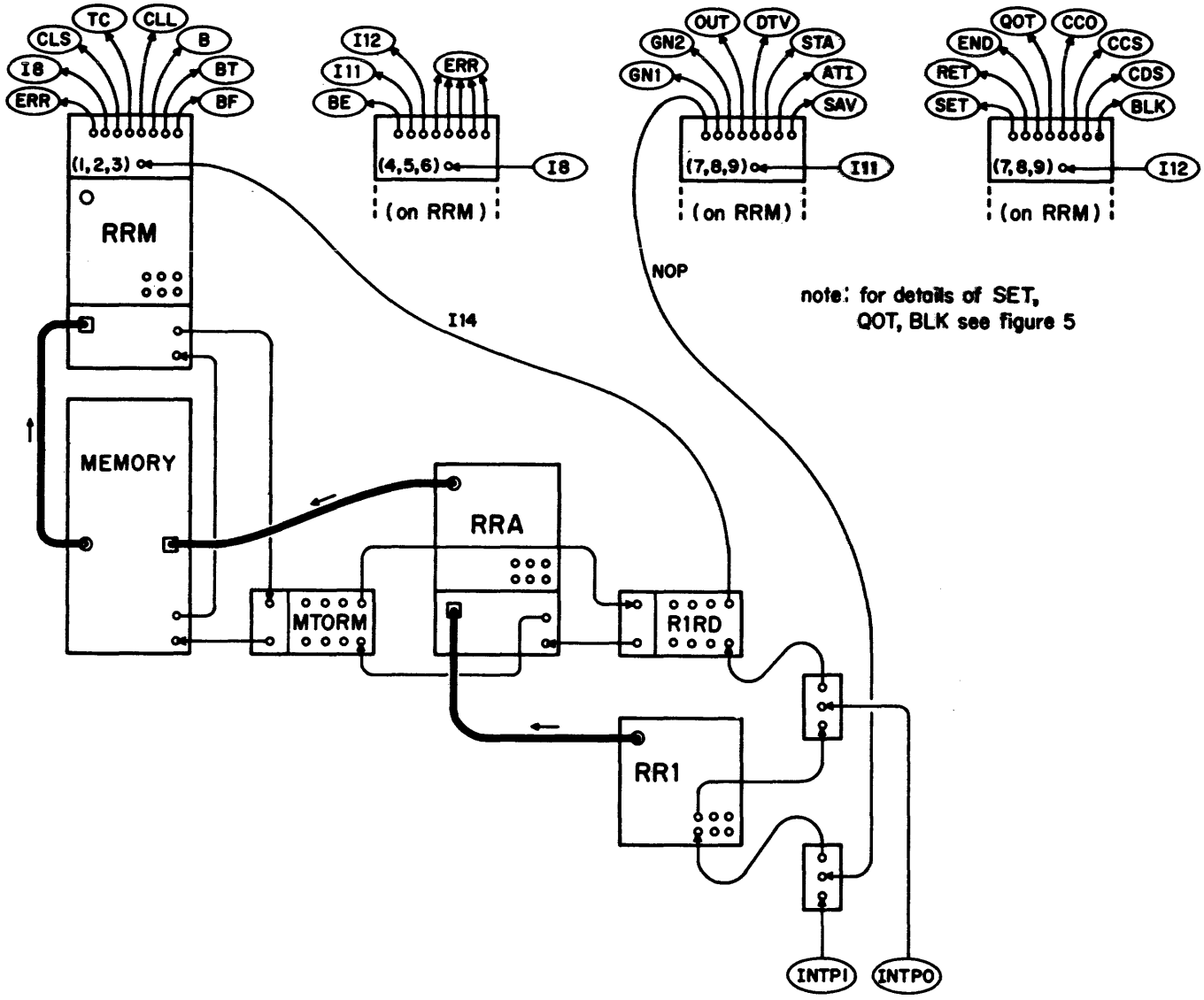


Figure 4 – Meta machine instruction decoding

one gated into the memory address register RRA. The subroutine MTORM is then used to put the contents of the location contained in RRA into the register RRM. This set of functions is grouped as a hardware subroutine because a number of other instructions also need to obtain the current input character, and hence need to execute exactly the same sequence of steps. After the return from routine R5RD, the control pulse continues on the line labeled B1. This line interrogates the detector unit on register RRM (labeled DD5) to see if the bit pattern of the current character matches the bit pattern for a blank. If the patterns do not match, control continues to line INTP1 thus advancing on to the next instruction. If the bit patterns do match, then another hardware subroutine is called by control line B2. Subroutine I13 increments the input string pointer register RR5. It also checks to see if the input string buffer is exhausted. If it is, a new card is read in and RR5 is reset to the initial buffer location. If it is not, RR5 is simply incremented by 1. On return from subroutine I13, control is merged with the original entry line, BLK. Thus, as long as black characters are found in the input they will be skipped. Control is not passed on to the next instruction until a non-blank current character is found.

Another entry in Figure 5 illustrates the connections for QOT. This instruction looks for a string quote in the input and sets the switch register to true if one is found. The first action is to use a call module to obtain the current input character. As in BLK, the hardware subroutine R5RD leaves the desired character in register RRM. On return from this subroutine, the control pulse continues on the line

labeled Q1. This line uses a detector DD2 to determine if the input character bit pattern matches the bit pattern of a string quote. If a match is found, the control line Q2 causes the current input character to be saved by calling the hardware subroutine INDX5. Subroutine I13 is then used to increment the input string pointer. The instruction SET is finally executed to set the switch register to true. If a string quote was not found, then the control line labeled T3 is activated. The contents of location 5 are read and stored in register RR5. This is done by putting the constant 5 into the memory address register, RRA, and then calling the MTORM subroutine. The result obtained in register RRM is then gated to register RR5. This restores the current input string pointer to a previously set value, allowing the input string to be scanned again. The control path then increments the switch register RR2, setting it to the value false. Merging with INTP1 then passes control on to the next instruction.

The third entry in Figure 5 is for SET. This instruction has two functions. It first clears the switch register RR2, by using the subroutine CLR2, setting it to the value true. It then writes the current input string pointer (the contents of register RR5) into memory location 5. SET, then, establishes the initial reference point for scanning a string. A failure in recognizing the input string will cause the input string pointer to be reset to the value it had at the last execution of the instruction SET.

The MS2 simulator was written to run on the IBM 360/50. As a matter of minor convenience, the bit patterns used for the characters in the meta machine are 8 bit EBCDIC, as used in the IBM 360 series computers.

This particular meta machine design requires the use of the following macromodular units:

43	data gates
41	call units
33	data branches
1	memory
11	registers
12	detectors
5	decoders
1	adder

plus a number of other miscellaneous units. It is quite interesting to note the interplay between the number of modules required and the length of program that must be executed. This machine design includes some rather powerful instructions, such as automatic table look up and push down stack operation. As a consequence, the entire basic compiler requires only 588 (decimal) locations. An alternative, a machine with less complex instructions, would obviously require fewer module units. However, the number of instructions would be increased as the balance between hardware and software shifted more towards the software side.

By shifting the emphasis even more to the hardware side, the compiler could be reduced further in size. Perhaps the extreme in this direction would be to eliminate the compiler activity completely. This could be done by designing the computer to accept, as direct machine instructions, statements in an algebraic language. Machines such as described by Bashkow, Sasson, and Kronfeld[5] could be implemented with macromodules. If this was done, a very interesting spectrum of machines, all based on a uniform hardware philosophy, would be available. All of the machines would be doing essentially the same job. Hardware complexity, however, would be a direct parameter effecting the ease, speed, economy, and general efficiency with which the machines would operate. Perhaps this approach would yield a more quantitative yardstick for evaluating new computer systems.

The design of a special purpose computer, the meta machine, has been presented. The intent has been twofold: (1) to demonstrate the applicability of the macromodular system to a non-numeric problem, and (2) to demonstrate the relative simplicity of the design function when the macromodular approach is used. An additional benefit is in the ease with which a macromodular machine may be modified after it has been assembled. It was also found that the macromodular concept forms the basis for a powerful simulation technique that is quite useful in itself.

REFERENCES

1 S M ORNSTEIN M J STUCKI W A CLARK
A Functional Description of Macromodules
Proceedings of the 1967 Spring Joint Computer Conference.

2 D V SCHORRE
Meta II A Syntax Oriented Compiler Writing Language
Proceedings of the 19th National ACM Conference, 1964

3 D K OPPENHEIM D P HAGGERTY
Meta 5: A Tool to Manipulate Strings of Data
Proceedings of the 21st National ACM Conference 1966

4 R A DAMMKOEHLER
A Macromodular Systems Simulator MS2
Proceedings of the 1967 Spring Joint Computer Conference

5 T R BASHKOW A SASSON A KRONFELD
Study of a Computer Directly Implementing an Algebraic Language
Columbia University Department of Electrical Engineering Technical Report No 87 January 1966

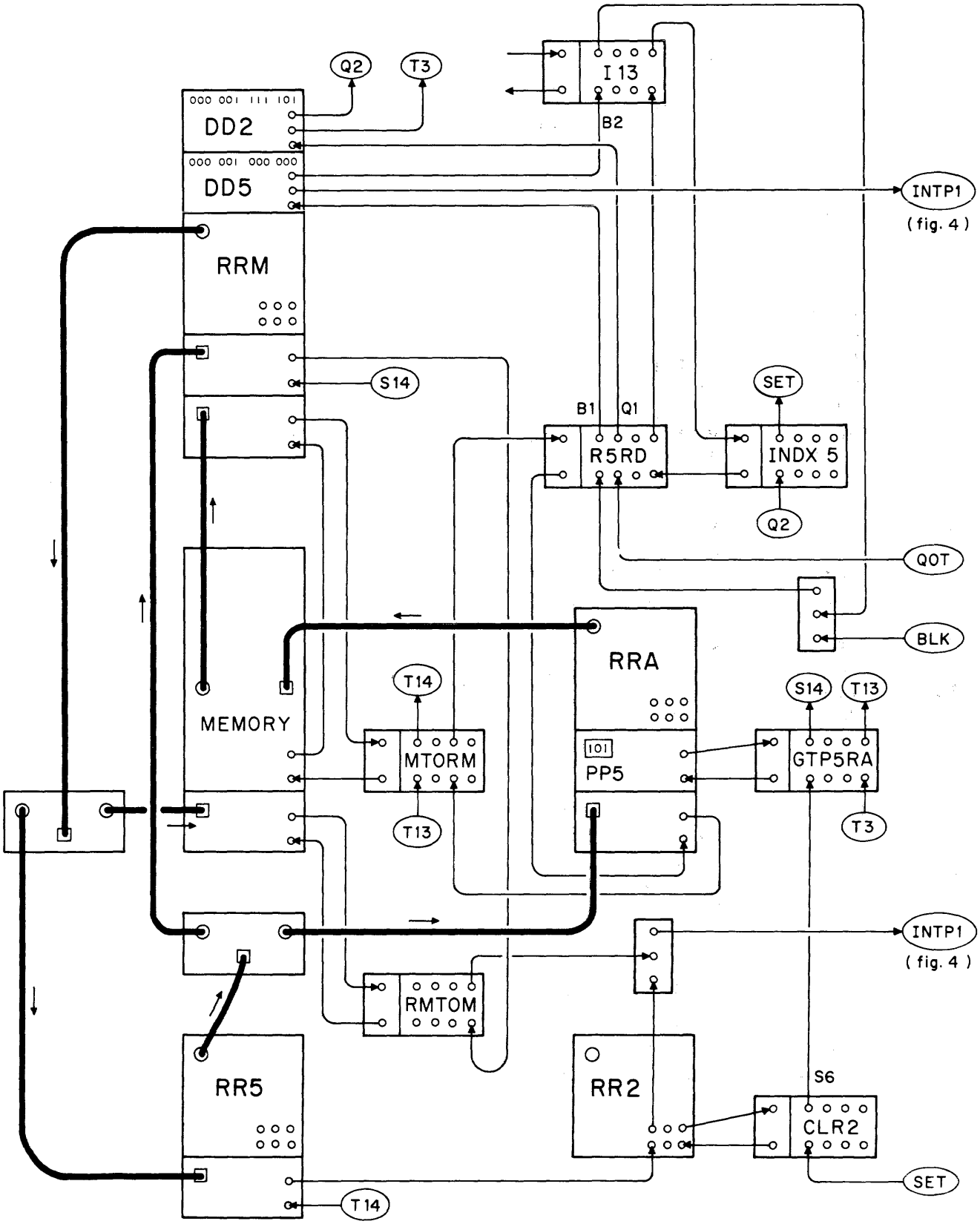


Figure 5 – Diagram for the instructions BLK,QOT,SET

```
BE*       DCALL,    DET2,     INTP1,    ERR'
BEGIN*    CLEAR,    RRA'
          GATE,     PP13U,    RR1'
          GATE,     RR1,      RRM'
          CALL,     CLR2,     B10'
  310*    CALL,     CLR3,     B14'
  314*    CALL,     RMTOM,    B3'
  33*     CALL,     P2RD,     B4'
  34*     CALL,     GTP3RA,   B5'
  35*     CALL,     RMTOM,    B15'
  315*    CALL,     GTP4RA,   B16'
  316*    CALL,     RMTOM,    B6'
  36*     CALL,     S8,       B13'
  313*    CALL,     I1,       B8'
  38*     CALL,     P1RD,     B9'
  39*     GATE,     RRM,      RR4'
          CALL,     DUMPR,    INTP0'
BF*       DCALL,    DET2,     INTP1,    B'
BLK*      CALL,     R5RD,     B1'
  31*     DETECT,   RRM,      DD5,      B2,       INTP1'
  32*     CALL,     I13,      BLK'
BT*       DCALL,    DET2,     B,        INTP1'
CC0*      CALL,     R5RD,     C1'
  C1*     CALL,     SETR3,    T4'
CCS*      CALL,     C2,       INTP1'
  C2*     CALL,     P4RDRA,   C3'
  C3*     CALL,     NEXTWD,   C4'
  C4*     CALL,     INDXM,    C23'
  C23*    CALL,     RMTOM,    C5'
  C5*     DECODE,   X<10>X<11>X<12>,RRM,ERR,C13,C14,C15,C16,C17,TRUE,TRUE'
  C13*    GATE,     JJ2,      RRM'
          MERGE,    C18'
  C14*    CALL,     NEXTWD,   C6'
  C6*     GATE,     JJ5,      RRM'
          MERGE,    C18'
  C15*    CALL,     NEXTWD,   C13'
  C16*    CALL,     RAM1,     C14'
  C17*    CALL,     RAM1,     C15'
  C18*    CALL,     RMTOM,    TRUE'
CDS*      CALL,     P4RDRA,   C8'
  C8*     CALL,     RAM4,     C9'
  C9*     CALL,     MTORM,    C10'
  C10*    GATE,     JJ3,      RRM'
          CALL,     RMTOM,    INTP1
CLL*      GATE,     RR1,      RR7'
          CALL,     GTRMR8,   C20'
  C20*    CALL,     PUSH,     C11'
  C11*    CALL,     CLR7,     C22'
  C22*    CALL,     PUSH,     C12'
  C12*    CALL,     PUSH,     C21'
  C21*    GATE,     RR8,      RRM'
          MERGE,    B'
CLR2*     CLEAR,    RR2'
CLR3*     CLEAR,    RR3'
CLR7*     CLEAR,    RR7'
CLS*      CALL,     GTRMR6,   CCS'
```

```
COMPA*      COMP,       RRA'
COMP4*      COMP,       RR4'
COMP6*      COMP,       RR6'
COMP7*      COMP,       RR7'
CR4RM*      COMPARE,    RR4,        RRM,        TRUE,       FALSE'
DET2*       DETECT,     RR2,        DD1,        TRUE,       FALSE'
DET9*       DETECT,     RR9,        DD6,        TRUE,       FALSE'
DTA*        CALL,       LOOK,       A1'
DUMPR*      OUTPUT,     RRA/RRM/RR1/RR2/RR3/RR4/RR5/RR6/RR7/RR8/RR9'
END*        CALL,       CLR7,       E9'
  E9*       CALL,       P4RDR6,     E2'
  E2*       CALL,       P3RD,       E7'
  E6*       CALL,       MTORM,      E3'
  E3*       OUTPUT,     RR6'
            OUTPUT,     RRM'
            COMPARE,    RRRA,       RR6,        E8,         E7'
  E7*       CALL,       RAM4,       E4'
  E4*       CALL,       R7M1,       E6'
  E8*       RETRN,      RR1'
ERR*        CALL,       DUMPR,      E5'
  E5*       RETRN,      RR1'
GTPORA*     GATE,       PP0,        RRA'
GTP1RA*     GATE,       PP1,        RRA'
GTP2RA*     GATE,       PP2,        RRA'
GTP3RA*     GATE,       PP3,        RRA'
GTP4RA*     GATE,       PP4,        RRA'
GTP5RA*     GATE,       PP5,        RRA'
GTRMRA*     GATE,       RRM,        RRA'
GTRMR6*     GATE,       RRM,        RR6'
GTRMR7*     GATE,       RRM,        RR7'
GTRMR8*     GATE,       RRM,        RR8'
GTR1RA*     GATE,       RR1,        RRA'
GTR4RA*     GATE,       RR4,        RRA'
GTR4R6*     GATE,       RR4,        RR6'
GTR5RA*     GATE,       RR5,        RRA'
GTR6RA*     GATE,       RR6,        RRA'
GTR6RM*     GATE,       RR6,        RRM'
GTR7RA*     GATE,       RR7,        RRA'
GTR7RM*     GATE,       RR7,        RRM'
GTR9RA*     GATE,       RR9,        RRA'
GN1*        CALL,       GTR4R6,     G1'
  G1*       GATE,       PP1,        RR8'
            GATE,       PP6,        RR9'
            MERGE,      G4'
GN2*        CALL,       GTR4R5,     G2'
  G2*       CALL,       R6M1,       G3'
  G3*       GATE,       PP2,        RR8'
            GATE,       PP7,        RR9'
            MERGE,      G4'
  G4*       CALL,       R6RDR7,     G5'
  G5*       DETECT,     RR7,        DD3,        G6,         G10'
  G6*       CALL,       R9RDR7,     G7'
  G7*       CALL,       INDX7,      G8'
  G8*       CALL,       R7WTR9,     G9'
  G9*       CALL,       R7WTR6,     G10'
  G10*      GATE,       RR8,        RR6'
```

```
          CALL,     C2,       G11'
  G11*    GATE,     RR7,      RR6'
          CALL,     C2,       G12'
  G12*    GATE,     JJ7,      RR6'
          CALL,     C2,       ATI'
INDXA*    INDEX,    RRA'
INDXM*    INDEX,    RRM'
INDX4*    INDEX,    RR4'
INDX5*    CALL,     R5RD,     I2'
  I2*     CALL,     GTRMR6,   I13'
  I13*    COMPARE,  PP130,    RR5,      I1,       I6'
  I1*     GATE,     PP10,     RR5'
          GATE,     PP10,     RRA'
          MERGE,    I5'
  I5*     INPUT,    RRM'
          CALL,     RMTOM,    I3'
  I3*     CALL,     INDXA,    I4'
  I4*     COMPARE,  PP130,    RRA,      TRUE,     I5'
  I6*     INDEX,    RR5'
INDX6*    INDEX,    RR6'
INDX7*    INDEX,    RR7'
INTP1*    INDEX,    RR1'
          MERGE,    INTP0'
INTP0*    CALL,     R1RD,     I7'
  I7*     CALL,     DUMPR,    I14'
  I14*    DECODE,   X<1>X<2>X<3>,RRM,ERR,I8,CLS,TC,CLL,B,BT,BF'
  I8*     DECODE,   X<4>X<5>X<6>,RRM, BE,I11,I12,ERR,ERR,ERR,ERR,ERR'
  I11*    DECODE,   X<7>X<8>X<9>,RRM,INTP1,GN1,GN2,OJT,OTV,STA,ATI,SAV'
  I12*    DECODE,   X<7>X<8>X<9>,RRM,SET,RET,END,QOT,CCO,CCS,CDS,BLK'
LOOK*     CALL,     P2RDR6,   L1'
  L1*     CALL,     P4RDR7,   L11'
  L11*    COMPARE,  RR6,      RR7,      S3,       L13'
  L12*    CALL,     GTR7RA,   S9'
  L13*    CLEAR,    RR9'
          CALL,     TEST,     L4'
  L4*     CALL,     TEST,     L5'
  L5*     CALL,     TEST,     L6'
  L6*     CALL,     INDX7,    L7'
  L7*     CALL,     INDX7,    L8'
  L8*     CALL,     INDX7,    L9'
  L9*     CALL,     R6M1,     L10'
  L10*    DCALL,    DET9,     L12,      L11'
MTORM*    READ,     RRM,      RRA'
NEXTWD*   CALL,     RAM1,     MTORM'
OUT*      CALL,     PORD,     O1'
  O1*     OUTPUT,   RRM'
          CALL,     INDXM,    O3'
  O3*     CALL,     RMTOM,    O2'
  O2*     OUTPUT,   RR3'
          CALL,     CLR3,     INTP1'
POP*      CALL,     GTR4RA,   P8'
  P8*     CALL,     MTORM,    P9'
  P9*     CALL,     GTRMR7,   P1'
  P1*     CALL,     P1RD,     P5'
  P5*     DCALL,    CR4RM,    ERR,      P4'
```

```
  P4*       CALL,     COMP4,    P2'
  P2*       CALL,     INDX4,    COMP4'
PUSH*       CALL,     INDX4,    P6'
  P6*       CALL,     P4RD,     P7'
  P7*       DCALL,    CR4RM,    ERR,      R7WTR4'
P0RD*       CALL,     GTP0RA,   MTORM'
P1RD*       CALL,     GTP1RA,   MTORM'
P2RD*       CALL,     GTP2RA,   MTORM'
P2RDR6*     CALL,     P2RD,     GTRMR6'
P3RD*       CALL,     GTP3RA,   MTORM'
P3RDR6*     CALL,     P3RD,     GTRMR6'
P4RD*       CALL,     GTP4RA,   MTORM'
P4RDRA*     CALL,     P4RD,     GTRMRA'
P4RDR6*     CALL,     P4RD,     GTRMR6'
P4RDR7*     CALL,     P4RD,     GTRMR7'
QDT*        CALL,     R5RD,     Q1'
  Q1*       DETECT,   RRM,      DD2,      Q2,       T3'
  Q2*       CALL,     INDX5,    SET'
RAM1*       CALL,     COMPA,    R10'
 R10*       CALL,     INDXA,    COMPA'
RAM4*       CALL,     COMPA,    R11'
 R11*       CALL,     INDXA,    R12'
 R12*       CALL,     INDXA,    R13'
 R13*       CALL,     INDXA,    R10'
RET*        CALL,     POP,      R1'
 R1*        CALL,     POP,      R2'
 R2*        CALL,     POP,      R3'
 R3*        GATE,     RR7,      RR1'
            MERGE,    INTP1'
RMTOM*      WRITE,    RRM,      RRA'
R1RD*       CALL,     GTR1RA,   MTORM'
R5RD*       CALL,     GTR5RA,   MTORM'
R6M1*       CALL,     COMP6,    R4'
 R4*        CALL,     INDX6,    COMP6'
R6M4*       CALL,     COMP6,    R14'
 R14*       CALL,     INDX6,    R15'
 R15*       CALL,     INDX6,    R16'
 R16*       CALL,     INDX6,    R4'
R6RD*       CALL,     GTR6RA,   MTORM'
R6RDR7*     CALL,     R6RD,     GTRMR7'
R6RDR8*     CALL,     R6RD,     GTRMR8'
R6WT*       CALL,     GTR6RM,   RMTOM'
R6WTP3*     CALL,     GTP3RA,   R6WT'
R6WTP4*     CALL,     GTP4RA,   R6WT'
R7M1*       CALL,     COMP7,    R5'
 R5*        CALL,     INDX7,    COMP7'
R7RD*       CALL,     GTR7RA,   MTORM'
R7WT*       CALL,     GTR7RM,   RMTOM'
R7WTR4*     CALL,     GTR4RA,   R7WT'
R7WTR6*     CALL,     GTR6RA,   R7WT'
R7WTR9*     CALL,     GTR9RA,   R7WT'
R9RD*       CALL,     GTR9RA,   MTORM'
R9RDR7*     CALL,     R9RD,     GTRMR7'
SAV*        CALL,     S4,       S1'
 S1*        CALL,     S5,       INTP1'
```

```
  S4*       CALL,     P3RDR6,   S2'
  S2*       CALL,     R6M4,     R6WTP3'
  S5*       CALL,     P4RDR6,   S3'
  S3*       CALL,     R6M4,     S7'
  S7*       CALL,     R6WTP4,   S8'
  S8*       CALL,     GTRMRA,   S9'
  S9*       CLEAR,    RRM'
            CALL,     S13,      S10'
  S10*      CALL,     S13,      S11'
  S11*      CALL,     S13,      S13'
  S13*      CALL,     RAM1,     RMTOM'
SET*        CALL,     CLR2,     S6'
  S6*       CALL,     GTP5RA,   S14'
  S14*      GATE,     RR5,      RRM'
            CALL,     RMTOM,    INTP1'
SETR3*      BITSET,   RRM,      RR3'
SHFTM*      SHFTRZ,   RRM'
STA*        CALL,     LOOK,     A3'
TC*         DCALL,    DET2,     T1,       INTP1'
  T1*       GATE,     JJ6,      RR7'
            CALL,     R5RD,     T2'
  T2*       COMPARE,  RRM,      RR7,      T4,       T3'
  T4*       CALL,     INDX5,    INTP1'
  T3*       CALL,     GTP5RA,   T13'
  T13*      CALL,     MTORM,    T14'
  T14*      GATE,     RRM,      RR5'
            INDEX,    RR2'
            MERGE,    INTP1'
TEST*       CALL,     R6M1,     T6'
  T6*       CALL,     R7M1,     T7'
  T7*       DCALL,    DET9,     T8,       TRUE'
  T8*       CALL,     R6RDR8,   T9'
  T9*       CALL,     R7RD,     T10'
  T10*      CALL,     DUMPR,    T12'
  T12*      COMPARE,  RR8,      RRM,      TRUE,     T11'
  T11*      INDEX,    RR9'
WTR6*       CALL,     GTR6RA,   RMTOM'
```

The CHASM: a macromodular computer for analyzing neuron models*

by CHARLES E. MOLNAR, SEVERO M. ORNSTEIN and ANTHARVEDI ANNÉ
Washington University
St. Louis, Missouri

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense through contract SD-302 and by the Division of Research Facilities and Resources of the National Institutes of Health through grant FR-00218.

INTRODUCTION

The CHASM is a "fixed plus variable"[1] computer system composed of a specialized marcomodular[2,3] processor coupled to a LINC[4]. The CHASM is designed to carry out the analysis of a Markov process model for the time patterns of spike discharges of neurons. The fixed portion of the CHASM computes certain probabilities associated with the Markov process model. The LINC provides input and output for the fixed portion and controls the execution of the calculations. The bulk of the calculations is efficiently carried out by the fixed portion. The more irregular operations, which occur less frequently, are carried out by the LINC.

The neuron model

The neuron model which the CHASM is designed to analyze was constructed to interpret the temporal patterns of single neuron spike discharges, observed through a microelectrode in the cochlear nucleus of the cat,[5] in terms of commonly accepted concepts of synaptic mechanisms.[6] Models similar to the present model have been used by various workers[7,8] to account for neural spike discharge patterns. The methods previously used to analyze these models have consisted primarily of some form of Monte-Carlo simulation, although Stein[8] has presented some analytical results. The method of solution used in the CHASM was first implemented in a less general form by means of a LINC program. Although the results obtained were satisfactory, the computing time was excessive for all but the simplest cases.

A detailed description of the earlier form of the model, as well as the rationalization of its form and some examples of its application to the cochlear nucleus, are given by Molnar.[9] The generalizations incorporated in the present form of the model are direct extensions of the ideas in the earlier model. These generalizations permit us to calculate the response of the model to time-varying inputs, as well as steady-state inputs, and also make it possible to simulate more closely the known features of the neural mechanisms on which the model is based.

We assume that the state of the model neuron is represented by a single continuous random variable which we shall call d, the depolarization of the neuron. The neuron receives inputs in the form of discrete events which occur at times given by a Poisson process with a specified time-dependent rate parameter, ρ (t).[10] The occurrence of an input event causes a discontinuous jump in the value of the depolarization with the jump size dependent upon the value of d. In the absence of input events, d undergoes a continuous change in value whose rate depends on the value of d.

The generation of an output event by the model, which we take as analogous to the spike discharge of a real neuron, occurs whenever the value of d exceeds a fixed threshold value T.

It can easily be seen that the random variable d is the state variable of a Markov process with a time-dependent parameter ρ (t). Figure 1 illustrates a simple example of a Markov process of this type. In this example, the jumps are of uniform size and the continuous transition is a linear decay.

We are particularly interested in obtaining two different results from this Markov process model. First,

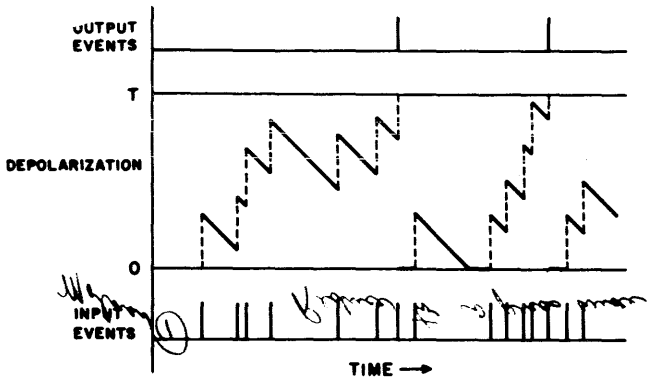


Figure 1 – An example of a Markov process with continuous and jump transitions

we wish to obtain the first passage time for the depolarization from some initial condition to the first crossing of the threshold value T. Second, we are interested in the density of threshold crossings as a function of time. The first case is analogous to the spike interval histogram, and the second to the post-stimulus time histogram.[11] These two statistics are commonly used in the neurophysiological literature to represent experimental results from studies of neuron spike discharge patterns.

The method used for the solution of the Markov process model is recursive calculation of the probability distribution of the depolarization at time $t+\Delta t$ from the previously calculated probability distribution for time t. Let us define $F_k(m)$ as the probability that the depolarization is less than $(m+1)\Delta x$ for $t=k\Delta t$, where m is an integer 0, 1, 2 ... 1023, and k is an integer 0, 1, 2 ... We shall henceforth choose the value of Δx such that $\Delta x = \frac{1024}{T}$. We will not discuss here in detail the errors that are introduced by making these discrete approximations to the original continuous problem, but it can be demonstrated that they are acceptably small for the purposes of the intended study.

We thus wish to calculate the function $F_{k+1}(m)$ from the known function $F_k(m)$. In order to do this, we must define two additional functions which characterize the continuous and jump transition mechanisms. The continuous transition mechanism, which may be regarded as decay of the depolarization towards zero, is described by the "backwards" function u_m. If the value of the depolarization at time $(k+1)\Delta t$ is $m\Delta x$, and no jumps took place in the interval $k\Delta t \leq t < (k+1)\Delta t$, then $u^m\Delta x$ is the value that the depolarization had at time $k\Delta t$. Likewise, we define $w_m^{\Delta x}$ as the value that the depolarization had at time $k\Delta t$, given that its value at time $(k+1)\Delta t$ is $m\Delta x$, and one jump took place during the time interval $k\Delta t \leq t < (k+1)\Delta t$.

To illustrate the meaning of the functions u_m and w_m, we consider the special case of the model which has received the most study. In this case, the jumps are of constant size $R\Delta x$, where we shall take R as an integer; the decay of depolarization follows an exponential curve

$$d(t) = d(t_0)\, e^{-\lambda\, t-to}$$

in the absence of any jumps. For this case,

$$\begin{aligned} w_m &= m - R \;,\; m-R \geq 0 \\ &= 0 \quad\;\; ,\; m-R < 0. \end{aligned}$$

Likewise, u_m is given, for this special case, by:

$$\begin{aligned} u^m &= (m)\, e^{\lambda\Delta t} \;,\; (m)\, e^{\lambda\Delta t} < 1024 \\ &= 0 \quad\;\; ,\; (m)\, e^{\lambda\Delta t} \geq 1024 \end{aligned}$$

Note that although the argument of u_m is an integer, its value need not be an integer. In other examples, the value of w_m may also be non-integer.

If we assume that the length of the time interval Δt is sufficiently short that the probability of more than one jump in an interval $k\Delta t \leq t < (k+1)\Delta t$ can be neglected, and take $p_k = \Delta t \rho(k\Delta t)$ as the probability of one jump occurring in that interval, we can write the following equation:

$$F_{k+1}(m) = (1-p_k)F_k(u_m) + p_k\, F_k\,(w_m). \quad (1)$$

This equation can be developed from the second integro-differential equation of Feller[12], which he derived in studies of a more general class of Markov processes. Alternatively, we can give the following heuristic justification of Equation (1).

If we assume that u_m is a monotonic function of m, and that no jump has taken place in the interval $k\Delta t \leq t < (k+1)\Delta t$, the value of u_m partitions the range of possible values of x_k in such a way that:

Prob. $\{x_{k+1} < (m+1)\Delta x\}$ = Prob. $\{x_k < u_m+1)\Delta x\}$, $F_{k+1}(m) = F_k(u_m)$.

Likewise, if we assume that w_m is a monotonic function of m, and that one jump took place in the time interval preceding $t = (k+1)\Delta t$, then the value of w_m partitions the range of possible values of x_k in such a way that:

Prob. $\{x_{k+1} < (m+1)\Delta x\}$ = Prob. $\{x_k < cu_m+1)\Delta x\}$, or $F_{k+1}(m) = F_k(u_m)$.

Using the theorem of total probability, and noting that the probability of one jump is p_k and of no jumps is $(1-p_k)$, we can directly write equation 1.

In the CHASM, the functions $F_k(m)$, u_m and w_m are each stored as 24-bit binary numbers in 1024

word tables. The result, $F_{k+1}(m)$, of a single recursion is stored in a fourth table of identical size. A map of the CHASM memory is shown in Figure 2.

The values of $F_k(m)$ are stored as positive 24 bit fractions, with the leftmost bit, serving as the sign Fgure 2) contains $F_k(m)$ for odd values of k, and Memory Table II (see Figure 2) contains $F_k(m)$ for even values of k. During a given iteration of the calculation of $F_{k+1}(m)$ the values needed for the right side of Equation (1) are obtained from one table, and the results stored in the other. Since $F_k(m)$ is a cumulative probability distribution, the maximum value of any entry is 1.0. This is approximated by the octal value 3777 7777. The values of u_m and w_m are stored with the integer part in the left twelve bits and the one's complement of the fractional part in the right twelve bits of the memory word.

The calculation required for each interval of time consists, in principal, of evaluating Equation (1) for m = 0, 1 . . . 1023, but the actual algorithm used in CHASM is expressed in a slightly different form. As noted earlier, the tabulated functions u_m and w_m have integer valued *arguments*, but may take on fractional *values*. In Equation (1), the values of u_m and w_m appear as arguments of $F_k(u_m)$ and $F_k(w_m)$, which have previously been defined only for integer arguments. Hence, we must define what is meant by the function $F_k(x)$ for non-integer values of x.

If we assume that $F_k(x)$ is piecewise linear between adjacent integer values of x, then $F_k(u_m)$ and $F_k(w_m)$ can be obtained by interpolation. Defining $(u_m)_i$ as the integer part, and $(u_m)_f$ as the fractional part of the value of u_m, we have:

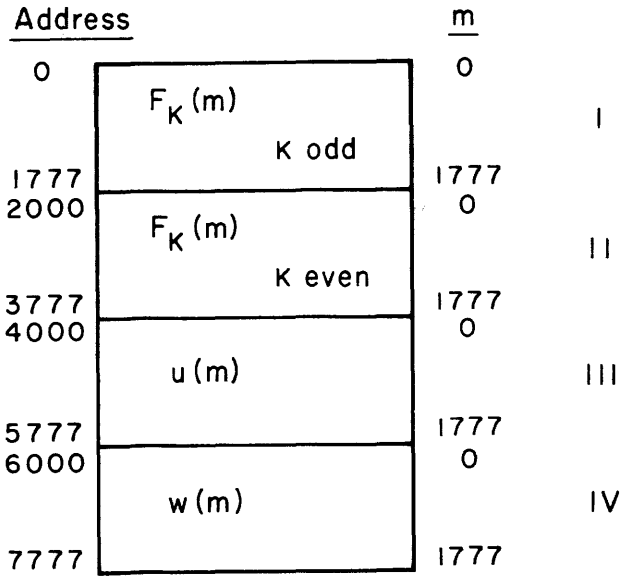


Figure 2—CHASM memory map

$$F_k(u_m) = F_k\left[(u_m)_i\right] + (u_m)_f\left\{F_k\left[(u_m)_i+1\right] - F_k\left[(u_m)_i\right]\right\}$$

This can be rewritten as:

$$F_k(u_m) = -\left[1-(u_m)_f\right]\left\{F_k\left[(u_m)_i+1\right] - F_k\left[(u_m)_i\right]\right\} + F_k\left[(u_m)_i+1\right] \quad (2)$$

By identical means:

$$F_k(w_m) = -\left[1-(w_m)_f\right]\left\{F_k\left[(w_m)_i+1\right] - F_k\left[(w_m)_i\right]\right\} + F_k\left[(w_m)_i+1\right] \quad (3)$$

Equations (2) and (3) give the values of $F_k(u_m)$ and $F_k(_m)$ needed to evaluate Equation (1).

The evaluation of both Equation (2) and Equation (3) is carried out in the CHASM by the same subsequence, which we shall call the Interpolate subsequence.

Figure 3 shows the complete data network of the CHASM, which identifies the registers, gates, data paths, adders, etc., that store, transfer and modify data. The two "accumulators", A and B, are each made up of two register macromodules coupled together to form one 24 bit register, and their gates, adders and shifters are likewise made up of coupled pairs of macromodules. The registers S and M are single macromodules used to store memory addressing information.

We assume that the calculation of $F_0(u_m)$ is in progress, that bits 0 – 9 of S contain the current value of m, and that $S_{11} = 1$ and $S_{10} = 0$.

The flow diagram in Figure 4 shows the sequence of operations carried out by the Interpolate subsequence in evaluating Equation (2). To the right of the flow diagram, the result of each step is shown in the form that is used in Equation (2). Reference to the data network in Figure 3, and the memory map of Figure 2 should clarify the interpretation of the flow diagram. Calculation of $F_0(w_m)$ is carried out if the subsequence is entered with $S_{11} = 1$ and $S_{10} = 1$. The value of $F_0(u_m)$ is left in register B at the conclusion of the subsequence. Step 5 of the flow diagram is not, strictly speaking, part of the calculation, but is needed to generate the proper memory address references to the tables containing $F_k(m)$.

Examining the flow diagram, we note that some of the steps in the calculation of $F_k(u_m)$ can be realized directly by single operations with macromodules, but other steps do not correspond to primitive macromodules. Just as in writing programs for conventional

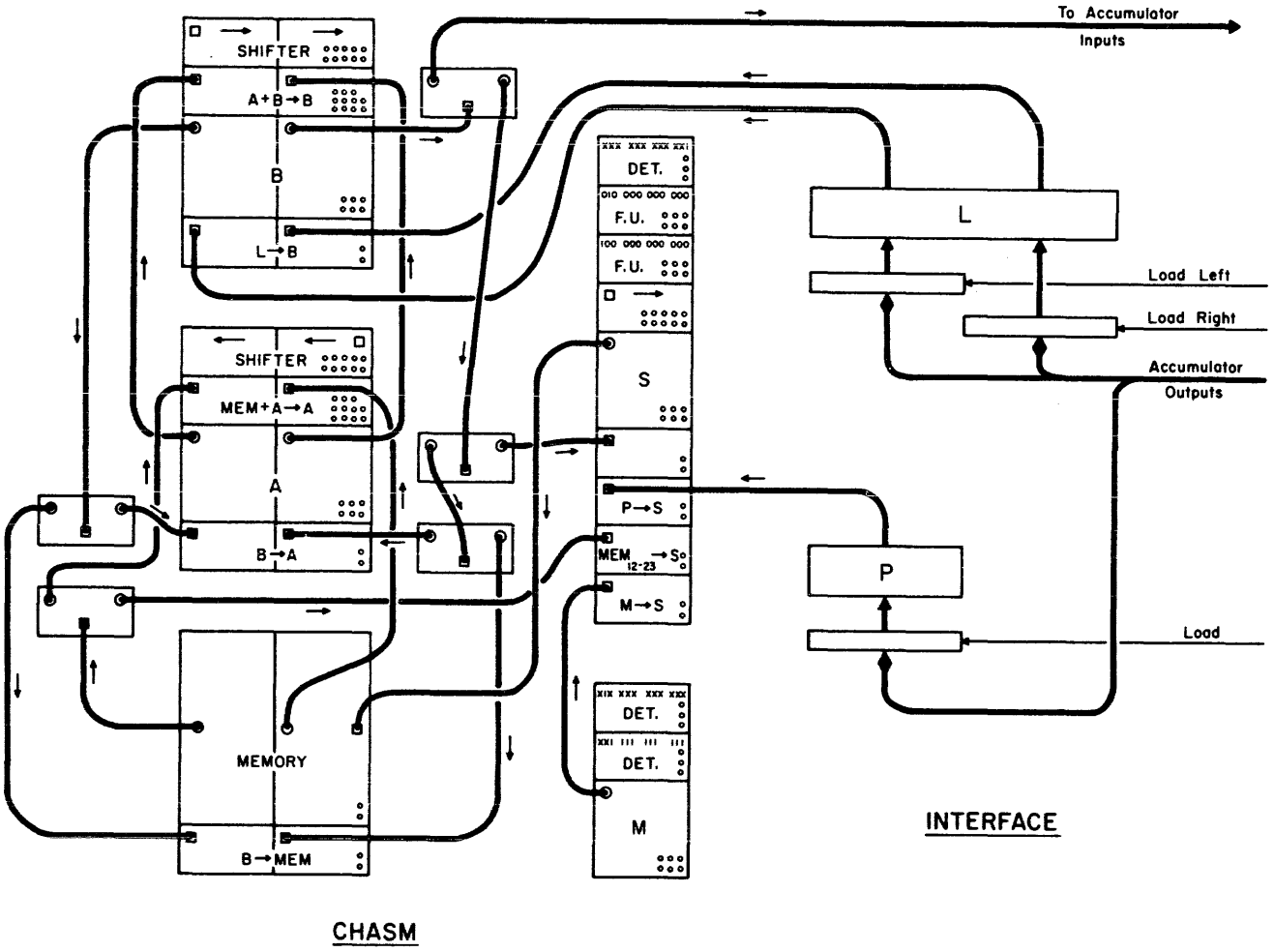


Figure 3—Data network and data interface

stored-program computers, subsequences of primitive operations are used to realize these more complex steps.

The operation MEM→A, which appears in steps (1) and (6), is composed of the three steps READ MEMORY, CLEAR A, and ADD MEMORY TO A. These three steps are realized by macromodular control connections shown in Figure 5(a). Similarly, the operation MEM-A→A is composed of the three primitive steps COMPLEMENT A, INDEX A, and ADD MEMORY TO A. The control cabling for this subsequence is shown in Figure 5(b).

The multiplication required in step (11) of the flow diagram in Figure 4 uses register S to hold the multiplier, B to hold the multiplicand, and A to accumulate the product. The macromodular control connections for the multiplication subsequence are shown in Figure 5(c). The detector macromodule senses the value of bit S_0, and by-passes the addition step if $S_0 = 0$. Counting of the number of iterations of the adding and shifting process is carried out by a pair of call units.

The complete macromodular implementation of the Interpolate subsequence is shown in Figure 6. The three subsequences, MEM→A, MEM-A→A, and S × A→B, each appear as subroutine within this subsequence. The first two are called twice within the Interpolate subsequence, and the S × A→B subsequence has been arranged as a subroutine in anticipation of further use by other parts of CHASM.

Figure 6 also shows, within the dotted line, some additional steps which are called the Normalize subsequence, and are always executed following the Interpolate subsequence. Their use will be explained later.

The evaluation of Equation (1) is carried out with similar techniques whose development we shall not describe in detail. The use of the M register for sequencing and addressing, and the interface with the LINC, however, do require further discussion.

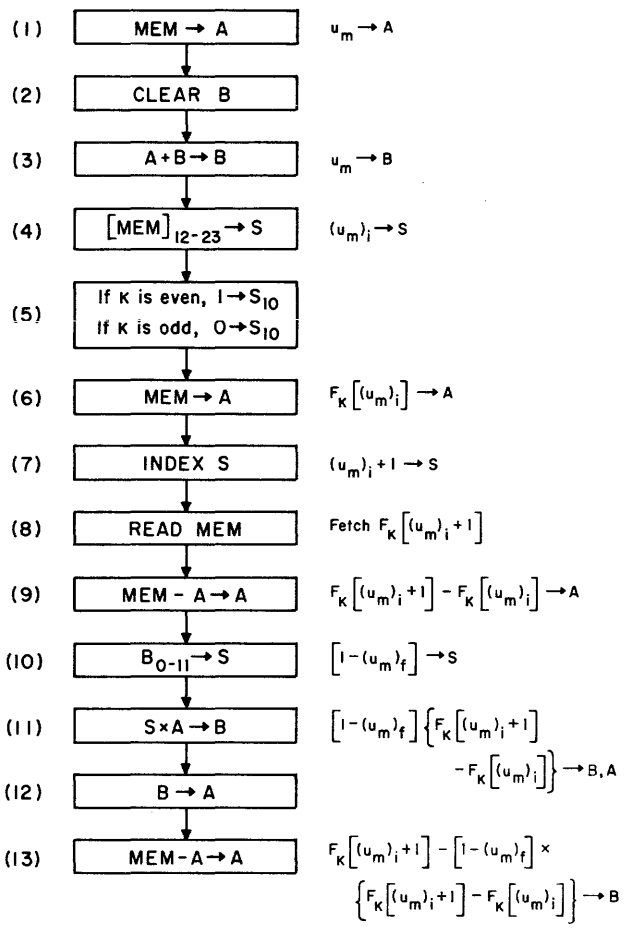


Figure 4 – Flow diagram of the interpolate subsequence

The LINC interface contains two registers, which we have designated as P and L. Register P is a twelve-bit register that is used to contain the current value of p_k. Register L is twenty-four bits in length, and is used to load information into the CHASM memory, such as the tabulated values of u_m and w_m and the initial probability distribution $F_0(m)$. These registers are loaded by the LINC under the control of a program in the LINC. Information is transferred from the CHASM to the LINC via a twelve-bit parallel path from the left half of register B to the LINC accumulator. The gating here is also under the control of the LINC.

The control interface between the LINC and the CHASM allows the LINC to initiate one of four sequences in the CHASM. The INITIALIZE and LOAD sequences are used to prepare CHASM for a series of calculations. The EXECUTE sequence is the major calculation sequence of the CHASM. The SHIFT B RIGHT sequence is used to move the left twelve bits of B into the right half of B so that they can be read into the LINC accumulator by the available gates.

Flow diagrams for these control sequences are shown in Figure 7. Following the completion of each sequence, a DONE signal is transmitted to the LINC, and the CHASM is inactive until another sequence of operations is initiated by the LINC.

Each evaluation of Equation (1) for a given pair of values of k and m requires nine references to core memory. The generation of the proper memory addresses for these references, as well as the storage of the current value of m, are functions of the M register. The sequence of memory references used is shown in Figure 8.

Before beginning a set of calculations, register M is set to the value zero by the INITIALIZE sequence. The current value of m is retained in the right ten bits, M_{0-9}, of the M register. Bit M_{10} is used to indicate whether the current value of k is odd or even, in the following way. The evaluation of Equation (1) is carried out for all 1024 possible values of m, beginning with m = 0 and ending with m = 1777_8. Following each iteration of Equation (1), register M is indexed (incremented by one), and bits M_{0-9} are examined. If they are all zero, a complete series of calculations for the given value of k has been completed, M_{0-9} are zero for the beginning of the next series of calculations for the next value of k, and bit M_{10} has been complemented by the carry out of M_9. Thus, bit M_{10} changes

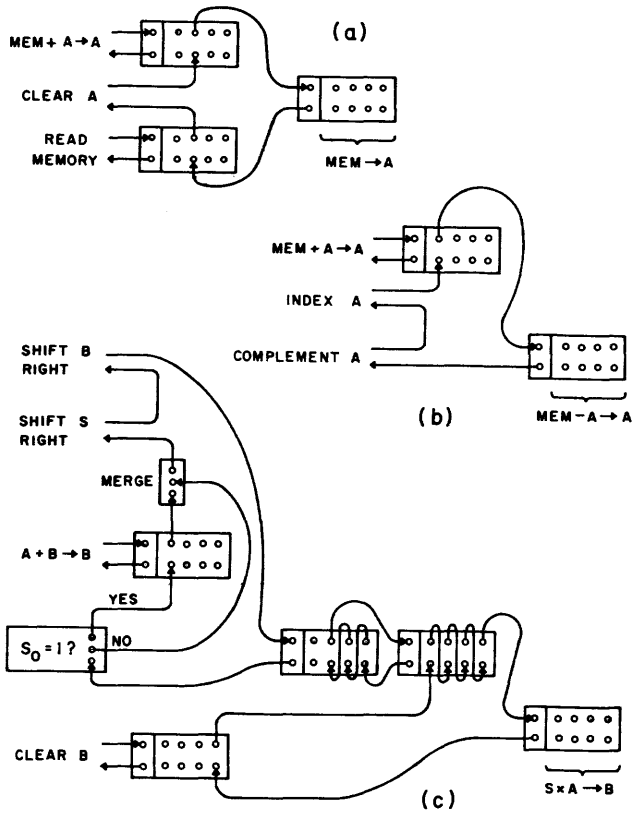


Figure 5 – Subroutines used by the interpolate subsequence

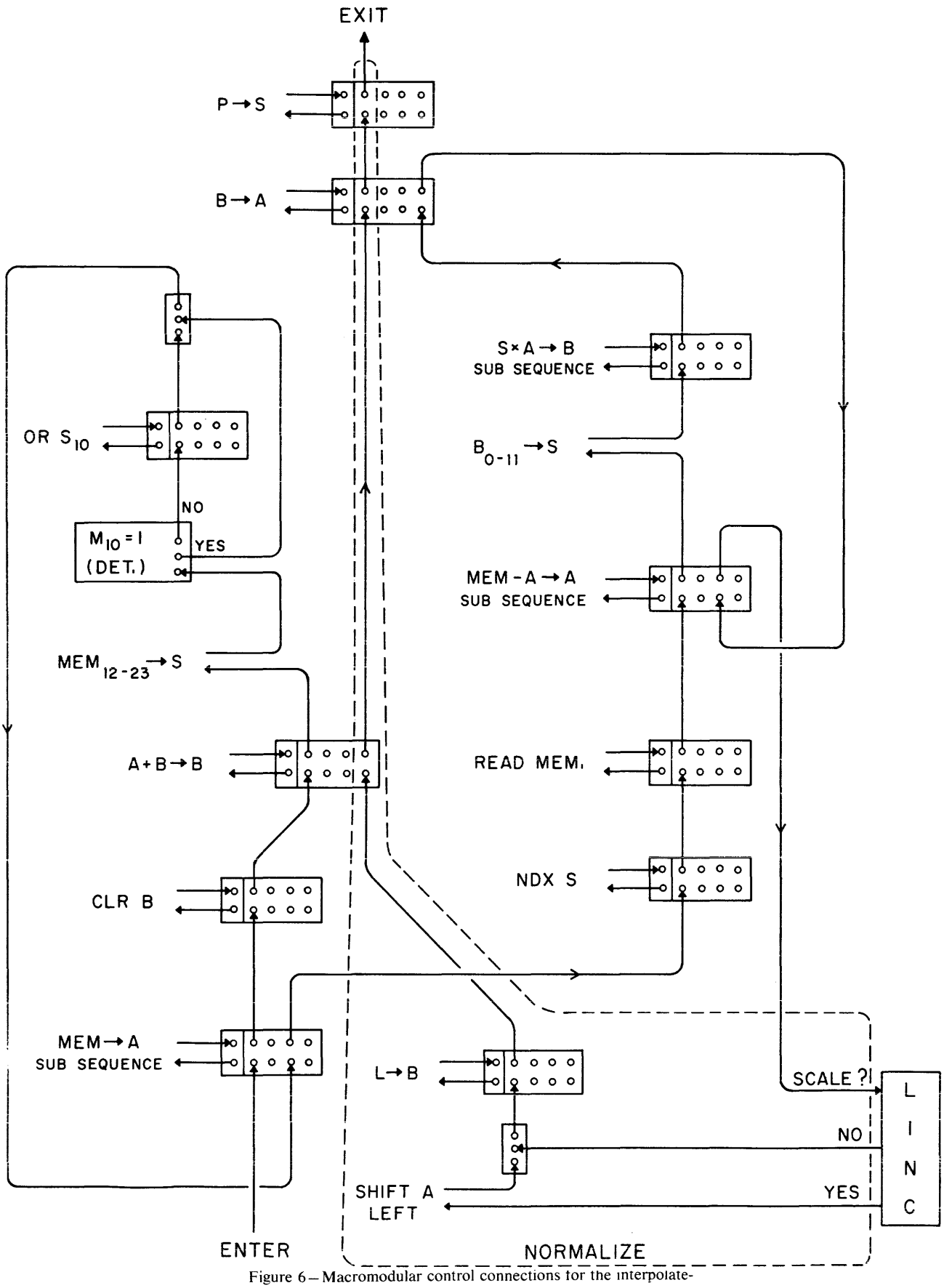


Figure 6—Macromodular control connections for the interpolate-normolize subroutine (I N S R)

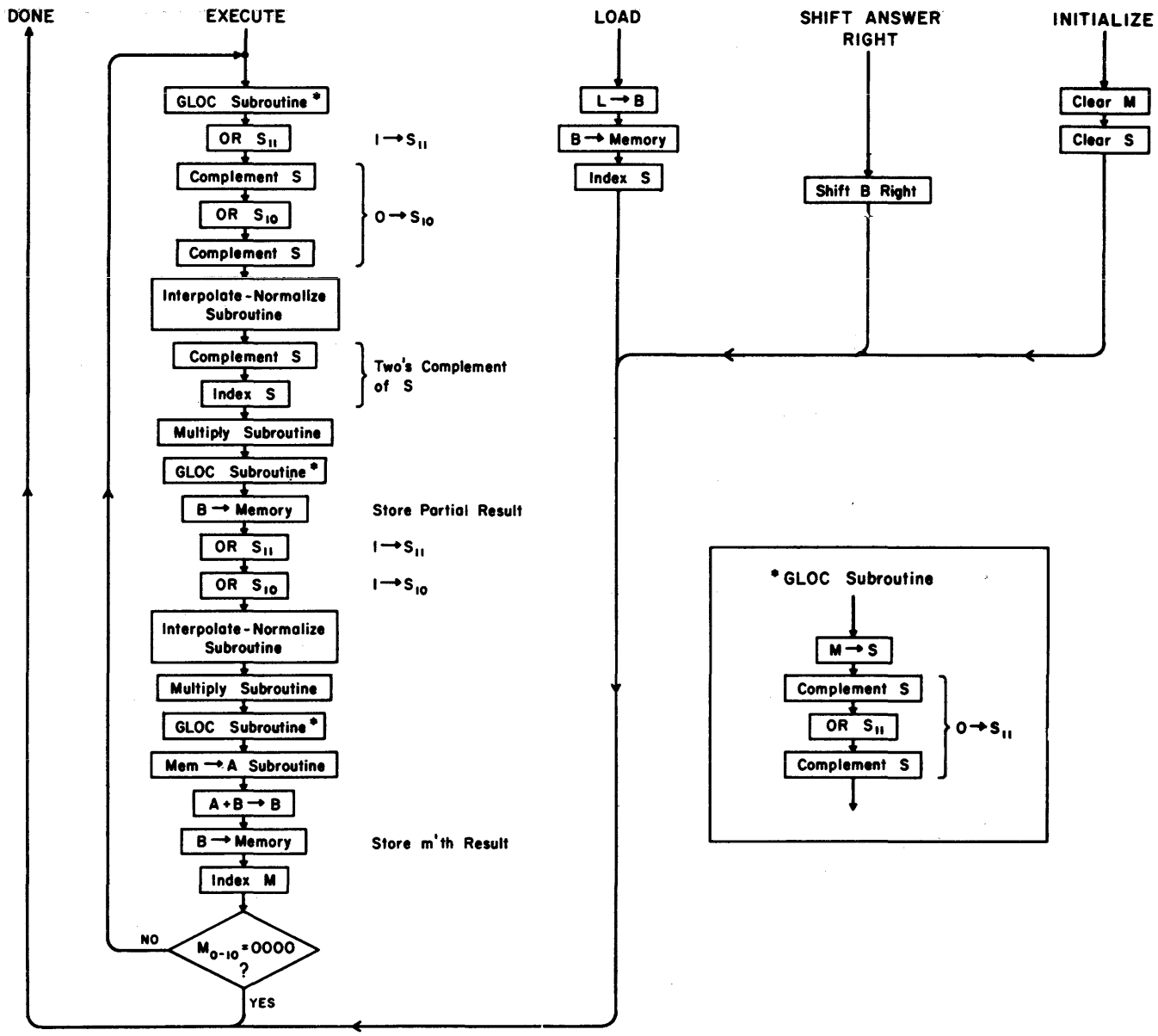


Figure 7 – Flow diagram of major programs

each time k changes, so that $M_{10} = 0$ signifies that k is even, and $M_{10} = 1$ signifies that k is odd. Bit M_{11}, which changes every second time that k is increased, is not used.

The subroutine GLOC sets S to the correct address for memory references number 4, 8 and 9 by copying M into S and clearing S_{11}. The correct addresses for references 2, 3, 6 and 7 are generated within the INTERPOLATE subsequence by the detector sensing M_{10} and the OR S_{10} operation. Note that bits S_{10-11} are originally derived from the tabulated values of u_m and w_m, which contain zeros in those positions, and S_{10} may subsequently be set to the value 1 if $M_{10} = 0$.

The major program in the CHASM is the EXECUTE program, which carries out all of the operations needed to calculate $F_{k+1}(m)$ from $F_k(m)$. The result of this calculation that is of greatest interest is the value of $F_{k+1}(1023)$, for the probability that the threshold was reached in the interval $k\Delta t \leq t < (k+1)\Delta t$ is given by $\{F_k(1023) - F_{k+1}(1023)\}$ The value of $F_{k+1}(1023)$, it can be observed, remains in B following the completion of the EXECUTE program, where it can be read into the LINC. The remainder of the calculation of the probability of a threshold crossing is carried out by a LINC program.

The LINC also chooses between calculation of the probability distribution of the interval between suc-

REF.	PURPOSE	ADDRESS	
(1)	Obtain u_m	4000 + m	
(2)	Obtain $F_K[(u_m)_i]$	$(u_m)_i$	(K odd)
		2000 + $(u_m)_i$	(K even)
(3)	Obtain $F_K[(u_m)_i + 1]$	$(u_m)_i + 1$	(K odd)
		2000 + $(u_m)_i + 1$	(K even)
(4)	Store $(1-p_K)\ F_K[u_m]$	2000 + m	(K odd)
		m	(K even)
(5)	Obtain w_m	6000 + m	
(6)	Obtain $F_K[(w_m)_i]$	$(w_m)_i$	(K odd)
		2000 + $(w_m)_i$	(K even)
(7)	Obtain $F_K[(w_m)_i + 1]$	$(w_m)_i + 1$	(K odd)
		2000 + $(w_m)_i + 1$	(K even)
(8)	Recover $(1-p_K)\ F_K[u_m]$	2000 + m	(K odd)
		m	(K even)
(9)	Store Result $F_{K+1}[m]$	2000 + m	(K odd)
		m	(K even)

Figure 8—List of memory references and addresses

cessive threshold crossings, and calculation of the density of threshold crossings. The first probability is analogous to the inter-spike interval histogram; the latter is analogous to the post-stimulus-time histogram. The former is simplest to calculate, as it is given by the values of successive differences between $F_k(1023)$ and $F_{k+1}(1023)$. The Markov process in this case is of the absorbing type, and the value of $F_k(1023)$ approaches zero, in general, as k increases. To maintain maximum precision in this calculation, provision is made for the value of $F_k(u_m)$ and $F_k(w_m)$ to be doubled after they are calculated by the Interpolate subsequence. This is done under control of the LINC whenever the previous value of $F_k(1023)$ is less than one-half.

The calculation of the density of threshold crossings is almost identical, except that we make the additional assumption that whenever the threshold is crossed the depolarization is reset to zero. This is represented in the calculation by adding a constant value, equal to $\{1 - F_k(1023)\}$, to each entry in the $F_k(m)$ table. This value is calculated by the LINC, and loaded into the L register, from where it is added to the value of $F_k(u_m)$ and $F_k(w_m)$.

Both of these steps are carried out by the Normalize subsequence. The Normalize subsequence directly follows the Interpolate subsequence; together they are identified and used as the Interpolate-Normalize Subroutine (INSR).

CHASM simulation

The design of the CHASM has been simulated on a LINC by Mr. R. Ellis, using a LINC macromodule simulator program written by him and Miss Carol Frankford. Although the scope of this simulation was limited by the small amount of memory available for the simulated CHASM memory, several design errors were discovered and corrected through the use of the simulator, and the resulting design was verified.

CONCLUSION

We anticipate that the CHASM will be a valuable tool for studies of spike discharge patterns of neurons in the cochlear nucleus of the cat. The increased speed of the CHASM over other available means of analyzing our model will permit us to explore a wider range of parameters of the model more fully than would otherwise be possible. Since the CHASM will be a relatively small and specialized computer, it should be feasible to use it intensively for periods of weeks to model results of particular animal experiments.

Some preliminary estimates of the relative cost of carrying out our modeling with the CHASM, as compared with the cost of using conventional computation center equipment and procedures, suggest that a cost advantage for the CHASM of an order of magnitude is possible if a large amount of calculation is to be carried out on this problem. This advantage is due to the relatively small cost of the equipment required for the CHASM as compared with the cost of the equipment which would be tied up during a similar calculation using a large machine.

As the design of the CHASM took less than two weeks to complete, we believe that macromodular computer systems for numerical analysis of relatively simple problems requiring a large amount of computation can be realized with an amount of effort comparable to that needed to write a program for a more conventional computer.

REFERENCES

1 G ESTRIN
Organization of computer systems: the fixed plus variable structure computer
Proceedings of the Western Joint Computer Conference 1960

2 W A CLARK S M ORNSTEIN M J STUCKI
A macromodular approach to computer design a preliminary report
Technical Report No 1 Washington University Computer Research Laboratory February 1966

3 S M ORNSTEIN M J STUCKI W A CLARK
A functional description of macromodules
Proceedings of the Spring Joint Computer Conference Atlantic City April 1967

4 W A CLARK C E MOLNAR
Computers in biomedical research
B Waxman and R Stacy Editors Vol II chap 2 Academic Press New York 1965

5 R R PHEIFFER N Y S KIANG
Spike discharge patterns of spontaneous and continuously stimulated activity in the cochlear nucleus of anesthetized cats
Biophys J 5, 301-316 1966

6 J C ECCLES
The Physiology of Synapses
Academic Press New York 1964

7 E E FETZ G L GERSTEIN
An RC model for spontaneous activity of single neurons
Quart Prog Report Mass Inst of Tech Research Lab Electron 71, 249-257 1963

8 R B STEIN
A theoretical analysis of neuron variability
Biophys J 5 173-194 1965

9 C E MOLNAR
Model for the convergence of inputs upon neurons in the cochlear nucleus
ScD Thesis Dept of Electrical Engineering Mass Inst of Tech 1966

10 E PARZEN
Stochastic processes
Holden Day San Francisco 1962

11 G L GERSTEIN N Y S KIANG
An approach to the quantitative analysis of electrophysiological data from single neurons
Biophysics J 1 15-28 1960

12 W FELLER
Zur theorie der stochastichen prozesse
Math Annalen 113 113-160 1936

Educational requirements for a student-subject matter interface

by KENNETH H. WODTKE
The Pennsylvania State University
University Park, Pennsylvania

INTRODUCTION

The problem of communicating knowledge, skills, and attitudes to students has concerned educators for centuries. Earlier debates centered around the use of lecture versus discussion methods, the use of the textbook, workbooks, etc. In general, these earlier "media" of instruction were passive display devices requiring relatively little response involvement on the part of the learner. Recent advances in communications technology have revived interest in the question of the media of instruction and have broadened the avenues through which the learner may come into contact with his subject. In general, the newer instructional display and response devices are "active" as opposed to earlier more passive media. They provide for the adaptation of the instructional materials to the characteristics of the individual learner as contrasted with earlier "static" display devices, and require active response involvement by the learner in the process of instruction. These devices allow the learner to *interact* overtly with his subject.

Paralleling the recent technological developments in the area of computers and communications systems, there has been an emerging technology of learning and instruction. A new breed of educational psychologist has emerged which is dedicated to the proposition that education can and should be a science and not an art. Armed with the tools of behaviorism and operationalism, he critically analyzes complex subject matters, behaviorally defines and evaluates instructional objectives, systematically arranges instructional experiences to optimize learning, and applies theoretical models to the process of instruction. It is not surprising that these investigators should also address themselves to the problem of the optimization of the modes of communication used in instruction.

In computer-assisted instruction (CAI) the term "student-subject matter interface" describes the devices such as two-way typewriter, cathode ray tube, slide projector, tape recorder, etc., through which the student interacts with the subject matter. The two major dimensions of the interface are its stimulus display capabilities and its response processing capabilities.* Stimulus display capability refers to the varieties of ways that a subject matter can be displayed to a learner through the interface. Response processing refers to the variety of student responses which can be detected by the interface and processed by a computer. The present paper will examine several classes of educational variables which determine the characteristics of an effective interface. The four categories of educational variables which must be considered in the design of the interface are the characteristics of the subject matter, the characteristics of the learner, the nature of the instructional process, and the objectives of instruction.

Subject matter characteristics

Perhaps one of the more obvious factors affecting the capabilities which must be available in an interface is the subject to be taught. Different subjects require different interface characteristics. The most extensive study of the relationships between subject matter characteristics and interface requirements

*The stimulus display and response processing capabilities of a CAI system are not solely dependent on the interface device. These capabilities are usually a joint function of both the interface, the computer, and the software. The lack of an important display or response processing feature may be due to a limitation in the interface device, the computer, the software, or some combination of several of these factors. Regardless of the source of the deficiency the net effect on students is the same. The lack of an audio capability may be just as serious whether it results from an inadequate recording and playback system, a limitation of the central computer, or a lack of available software. The purpose of this paper is not to analyze how the various display and response processing capabilities may be implemented technologically. This would be a task for the engineer and computer scientist. The present paper will be concerned with the educational variables which must be considered in establishing specifications for a student-subject matter interface.

has been made by Glaser, Ramage, and Lipson.[1] These investigators conducted an extensive analysis of the behaviors involved in learning three subjects: elementary mathematics, elementary reading, and elementary science. The analysis was based on an actual survey of instructional programs and curricula used in the schools. From this task analysis Glaser and his associates were able to draw up a set of specifications for an instructional environment to teach each of the subjects. A summary of their findings is shown in Table 1. The table shows the wide variety of interface requirements for elementary mathematics, reading, and science subjects. Science appears to provide the greatest challenge to the development of an instructional environment. For example, science instruction often requires the visual display of natural phenomena and simulation pictures. One can't help but note the obvious advantages of a closed circuit television capability as part of the interface for displaying scientific phenomena. Science instruction also appears to place rather sizable demands on the interface in the area of response processing, thus, there are times when the learner may need to manipulate controls, construct graphs and figures, etc.

Some of the requirements listed in Table I are easier to implement than others. For example, our experience at the Penn State CAI Laboratory indicates that it is a relatively simple matter to provide various types of manipulanda for the student at the

Table I

Stimulus and Response Requirements for the Instructional Environment in Elementary Mathematics, Reading, and Science

	MATHEMATICS	READING	SCIENCE
Visual Stimulus	*Numerals *Symbols *Drawings *Pictures *Words *Objects *Meaningful arrays	*Pictures *Symbols *Letters *Words	*Natural phenomena *Pictures and drawings *Simuluation pictures
Auditory Stimulus	*Sound patterns *Words (for young students)	*Words *Sounds for symbols	*Sound phenomena
Tactile Stimulus	Objects for manipulation	Letters (for young students	Natural phenomena-friction, heat, textures, etc.
Visual Display for Response	*Numeral *Figures *Symbols *Objects *Letters and words *Sequential choice display	*Letters *Pictures *Words	Manipulative controls *Sequential choice display
Auditory Response Detection	Words (for young students)	Phonemes Words Names of letters Words in context	Words (for young students)
Object Display and Manipulation	Numbers of objects Patterns of objects	Letters as objects (for young students)	Models and objects of science area
Symbol and Figure Construction	Geometric figures		*Graph construction *Figure construction
Sequential Stimuli	*Algorithm problems	*Reading in context	*Cycles *Natural sequences *Laboratory sequences
Time Factor Presentation	*Patterns in time	*Introduction of prompts with delayed response	*Time factor in natural phenomena

From Glaser, R., Ramage, W. W., and Lipson, J. I. The Interface Between Student and Subject Matter. Learning Research and Development Center, University of Pittsburg, Pittsburgh, Pa., 1964. pp. 177.

*Instructional functions which probably can be implemented in a CAI system within the foreseeable future.

instructional terminal. One course in audiology covering the anatomy of the ear provides the student with various plastic models of the ear and the skull. The student is taught to identify the anatomical parts of the ear both by reference to slides depicting the various parts of the auditory system and the actual models.

One conclusion seems evident from an examination of Table I. Existing interface capabilities cannot accommodate all of the requirements of these subject matters as they are presently taught in the schools. Furthermore, it is doubtful whether some of these capabilities (voice detection for example) will be feasible in the very near future. This observation is not surprising in view of the fact that most classroom instruction provides a wide variety of experiences for the learner. It must be remembered that Glaser's analysis represents the total spectrum of instructional experiences and few investigators expect CAI to reproduce all aspects of these experiences. The rather marked discrepancies between the requirements of Table I, and current interface capabilities reemphasizes the widely held view that CAI must be *supplementary* to existing instructional techniques. Mitzel[2] has pointed out that one of the most important problems to be solved in the development of CAI is the determination of the appropriate "mix" between computer-mediated instruction and components of instruction which are mediated by the teacher or by still other means such as educational television. It is likely that the nature of the "mix" will be determined in part by the kinds of instructional experiences which can be efficiently and effectively provided by the CAI interface. Those experiences which cannot be implemented via CAI must be provided by the teacher or by some other means.

What are the instructional functions which can feasibly be provided within the next generation of CAI systems? I have indicated my own conjectures with regard to this question by marking with an asterisk those instructional functions listed in Table I which I think can be implemented within a CAI system in the foreseeable future.

Work with high school and college level CAI courses at Penn State (Mitzel,[3] Mitzel and Brandon [4,5,6]) indicates that the level of the subject matter is also a critical variable in determining the requirements of the interface. Elementary mathematics presents one set of interface problems and higher mathematics another.

The nature of responses in a mathematics program are often such that the course author is willing to accept as correct a range of answers. In the course of solving a complex mathematics problem rounding errors may result in minor variations in the numerical answer. If the computer is programmed to require a perfect match of the student's answer with a prestored answer, answers which are essentially correct will be regarded as incorrect by the machine. As a solution to this problem, IBM[7] has developed a "limit function" as part of their *Coursewriter* language. Limit function allows an author to program the computer to accept all responses within certain specified limits as correct responses.

Another problem occurs in attempting to teach college mathematics using a typewriter terminal to display the course and process student responses. Mathematics frequently involves the display of relatively complex equations and symbolic configurations containing subscripts and superscripts. The typewriter is not efficient in dealing with such materials, and may actually interfere with learning when the student is confronted with the complex task of inputting symbolic material. The cathode ray tube display may alleviate this problem to some extent, but the problem of inputting symbolic material through a keyboard still remains. It is noteworthy that even the most experienced typists have difficulty in reproducing material containing many symbols and equations. IBM attempted to solve this problem by developing a mathematics typehead and a keyboard mask to aid the student in identifying the new character set. Our experience with this device at Penn State suggests that it is satisfactory for relatively short responses, but that it does not greatly simplify the inputting task for longer symbolic responses. IBM's discontinuation of the mathematics typehead for its 1050 communications system suggests that they may have had similar misgivings concerning this procedure.

Mathematics is not the only subject matter which presents special problems for the CAI interface. Several studies conducted at Penn State (Wodtke and Gilman,[8] Wodtke, Gilman, and Logan[9]), demonstrate that the typewriter is an inefficient interface for highly verbal subjects at the college level. When identical instructional programs were administered on-terminal and off-terminal there was an increase in instructional time of 25 per cent in the on-terminal group with no commensurate increase in learning. In a second study employing a still more highly verbal program, the increase in instructional time was 75 per cent for the on-terminal group with no significant difference in learning when compared to the off-terminal group. The largest portion of this time decrement is undoubtedly due to the slow type-out rate of the typewriter (approximately 120 words per minute)

which is substantially slower than the normal reading speed of the typical high school or college student.

In an area of research where variations in instruction typically produce only small gains in student achievement, a time loss of 25 per cent represents a substantial decrement. The time could be used to give students more practice, instruction on new material, or practice on transfer problems. In addition to the gains in student learning which might accrue from a more efficient use of instructional time, there are also economic considerations in the cost of computer time, tie-lines, and other "hidden" costs involved in the preparation of courses. All other things being equal, by employing an interface which would decrease instructional time by 25 per cent without reducing the amount learned, four students could be taught for every three taught by means of a typewriter interface.

From the college student's point of view, learning at a typewriter terminal is not self-paced instruction since he must slow down his normal rate of work. Pacing instruction below a student's optimal rate could produce boredom, negativism, and avoidance of CAI as an aid to learning. This is not an uncommon finding when the pace of classroom instruction by the lecture method is too slow for the brighter students. The advent of the cathode ray tube display device should speed up substantially the display of verbal information to students.

The results obtained with the typewriter interface at Penn State are not consistent with the results reported in several other studies. Grubb and Selfridge[10] compared the performance of college students taught descriptive statistics via CAI, conventional lectures, and programmed text. Those students taught via CAI took one-tenth as long and achieved twice as well on the final posttest as did the other two groups! These results are so spectacular that they demand replication by other investigators. Schurdak[11] found that students who learned Fortran programming via CAI saved 10 per cent in instructional time, and performed 10 per cent better than students using a standard text or a programmed text. These somewhat more modest results conform more to expectations than the phenomenal results reported by Grubb and Selfridge.

The nature of the subject matter is also an important determiner of the response processing requirements of an interface device. Relatively simple drill programs may not require very extensive response processing since most student responses are of the short answer variety. College-level tutorial and problem-simulation programs will ordinarily require a CAI system with partial-answer processing capability. Partial answer processing refers to the capability of the computer to search a student's response for the essential elements of the correct answer; to disregard minor character mismatches, spelling, and typing errors, to regard or disregard word order in a student's response depending on the nature of the problem; etc. In one preliminary study, Wodtke[12] examined the relationship between student performance in a CAI mathematics course, and the number of times a student entered a correct answer at the terminal which was regarded as incorrect by the computer because of a minor character mismatch. The correlation between student achievement in the course, and the number of mismatched correct answers was—.80! Thus, it is quite clear that inadequacies in the computer's ability to detect correct answers seriously interferes with student learning. It should be noted that detection of correct responses is not solely an interface problem, but is also a software problem. The partial answer processing capability in the IBM CAI system is provided in the *Coursewriter* author language.

Learner characteristics

The second class of variables which must be considered in the design of an effective student-subject matter interface are individual differences among the students. The effects of some individual difference variables on the interface are obvious, for example, different interface capabilities are required for young children as compared to college students or adults. Whereas auditory stimulus display capability may be "supplementary" at the adult level, it is absolutely essential in instruction with very young children who are still nonreaders. Auditory communication would be the primary means of communication with nonreading youngsters. Glaser, Ramage, and Lipson[1] point out that auditory response *detection* would also be an essential requirement for an interface in teaching young students some aspects of mathematics, reading, and science. For example, in reviewing an elementary reading curriculum, Glaser and his associates point out that the student must acquire the following competencies involving an oral response: (1) In learning sound-symbol correspondence, the student is asked to pronounce the sounds of individual letters when written on the board or to circle or write the appropriate letter when its sound is presented, and (2) At all stages the student is asked to read aloud stories using the words he has learned. It is probable that some instructional functions such as oral reading which require oral response detection will have to be delegated to the teacher, with CAI incorporating those functions which are feasible within the present technology.

Object manipulation is likely to be another important instructional experience for very young pupils, and for older students in some subject matters such as science. According to Piaget's[13] stage theory of human development, children pass through several stages of development. Early developmental stages are characterized by sensorimotor development and the manipulation of concrete objects. Later stages are characterized by the ability to operate in more abstract terms. According to this view, the manipulation of concrete objects would be an indispensable part of instruction for elementary school children.

Although there may be mechanical methods for providing experience in the manipulation of concrete objects (Glaser, Ramage, and Lipson[1] have described an electronic manipulation board which would be capable of detecting the identity and placement of objects located on its surface), within the present stage of technology, it would seem more efficient to delegate this function to the teacher.

The CAI interface must provide a maximum of flexibility in adapting display and response modes to differences in student aptitude and past achievement. An author should have the capability of speeding up or slowing down the flow of information to a student depending on the student's aptitude or progress through the course. The variation in the rate of presentation of instruction requires a time-out feature (now available on most CAI systems) which enables the author to regain control of the terminal in the case of a student whose response latencies are excessively long. Without this terminal control, it is difficult for an author working through his computer program to alter the pace of instruction.

The need for the interface to provide graphics and auditory display capabilities also depends on the past experiences and backgrounds of the students. The typical college undergraduate has high verbal ability. CAI for college students can rely heavily on communication by means of verbal material displayed via a typewriter or cathode ray tube. However, if one considers instruction for culturally disadvantaged students or students in vocational and technical education programs, much more reliance must be placed on relatively nonverbal media of instruction. Such students will probably require more graphic displays in the form of pictures, diagrams, and drawings depicting the concepts of the course. These students may have difficulty thinking in abstract verbal terms and may require much more actual manipulation of the objects of instruction. In addition, the relatively nonverbal student might show considerable improvement in learning when verbal instruction is supplemented with auditory communication. The problem of how to communicate concepts to learners who are handicapped in verbal communication skills is an important problem for research with direct implications for the design of CAI display devices.

One learner characteristic which has important implications for the determination of the appropriate "mix" between computer-mediated and teacher-mediated instruction is the student's need for affection, nurturance, and personal contact with a teacher. This variable would be particularly important during the early stages of learning with young children. It is well known that children vary considerably in their need for affection and contact with a teacher. The need seems to be particularly acute in less mature youngsters when they are initially confronted with the complexities of a difficult learning task. Increased teacher support and nurturance may be required during the early stages of the development of a complex skill such as learning to read. Children should be tested prior to instruction so that some decision can be made concerning their need for support and contact with a teacher. The instructional system must then provide the additional personal contact required by a given child.

The branching and decision-making capabilities of the computer are the most unique and potentially important characteristics of a CAI system. To the writer's knowledge, all current CAI systems provide the ability to store information concerning the student's learning history, and the ability to make decisions to branch the student to instructional material which is appropriately suited to the particular learning history. The decision-making capability of most CAI systems is one technological capability which far exceeds our present knowledge of the process of learning and instruction. Unfortunately, psychologists are hard put to know which characteristics of the learner (immediate response history, pattern of aptitudes, response latencies, etc.) are optimum for branching decisions, and until we discover the instructional experiences which are optimal for a student with a particular learning history, the full power of CAI for individualizing instruction will not be realized.

Characteristics of the instructional process

The nature of the instructional process must also be considered in the design of a student-subject matter interface. It will not be possible in the present paper to consider all of the instructional variables which may effect the efficiency of an interface; however, some of the variables which may have special relevance to interface design will be considered.

It is a commonly accepted principle in all theories of learning, that there must be contiguity of stimulus

and response for learning to occur. In common parlance, the principle of contiguity simply means that in order for a response to become associated with a stimulus, the response must occur either implicitly or explicitly in the presence of the stimulus. If a student is to learn the English equivalent of a German word, he must reproduce the English word (explicitly or implicitly) while *attending* to the German word. Hence, attention to the stimulus word or instructional display is a critical factor in instruction. It is therefore important to determine the characteristics of displays which produce a high degree of attention on the part of students. Perhaps the most relevant research on this question has been conducted by Berlyne.[14] Berlyne found that attention to a stimulus is heightened by the element of surprise, change, or novelty of the stimulus. When the same stimuli are presented in the same format over repeated presentations, attention to the task wanes, and motivation declines. The effects of novelty on behavior can be observed in any student working at a CAI terminal for the first time. The novelty of working with a machine which "talks back" has high attention holding value, at least during the early stages of instruction. The interface must be capable of providing enough variety of stimulation and novelty to sustain attention over an extended period of time. An interface which because of limited display and response processing capability provides for only very simple display and short answer response formats such as the multiple choice frame will soon lose interest for the learner.

Travers[15] and his colleagues have recently conducted an extensive review and research on various aspects of audiovisual information transmission. Travers' studies and those of Glaser, Ramage and Lipson[1] must be considered among the primary references in the field of interface design. Travers has re-examined the traditional view that the primary advantage of audiovisual presentations is in the realism they provide. Contrary to this traditional view Travers argues that increased realism may actually interfere with learning by providing too many irrelevant cues, thus, a simple line drawing may be more effective in communicating essential concepts to learners than a more realistic picture. Travers concludes:

> *First, the evidence points to the conclusion that simplification results in improved learning. This seems to be generally true regardless of the nature of the presentation—whether it is pictorial or verbal. This raises interesting problems, for the simplification of audiovisual materials is that they provide a degree of pictorial content will generally result in a less realistic presentation than a presentation which is close to the life situation. The argument which is commonly given in favor of realism which other procedures do not. The realism provided may not be entirely an advantage and may interfere with the transmittal of information. The problem of simplifying realistic situations so that the situations retain information essential for permitting the learner to respond effectively at some later time to other realistic situations, is an important one!*[15] *(pp.2.110-2.111).*

The limited capacity of a cathode ray tube for displaying a high degree of realism may not be such a deficiency after all.

A more recent experiment suggests that the above generalization may have to be qualified in several respects. Although simplified displays may facilitate transmission of information they do not appear to facilitate transfer or application of what has been learned as well as more realistic displays. Overing and Travers[16] found that students who were taught the principle of refraction of light in water by means of realistic demonstrations were better able to apply the principle in attempting to hit an underwater target than students who were taught by means of simplified line drawings. This study suggests that the primary advantage of realistic displays may be in helping the student to apply what he has learned later on in realistic problem solving situations.

Travers[15] and Van Mondfrans and Travers[17] have also conducted research on the relative efficiency of information transmission through the auditory or visual senses, or some combination of the two senses. In general, their results suggest that instruction involving the visual modality is superior to auditory instruction when the auditory presentation produces some ambiguity in the information transmitted. Thus, the auditory modality was distinctly inferior to all other modality combinations in the learning of a list of nonsense syllables, but no differences between modalities were obtained when meaningful materials were learned. The auditory transmission of a nonsense term produces considerably more ambiguity of interpretation than the auditory transmission of a meaningful word. These results suggest that the use of the visual modality will be particularly effective when the task is to learn new associations such as in learning a foreign language.

Another relevant consideration in interface design is the capacity of the human being for processing information. Travers[15] favors a single channel theory of information processing. According to this view,

the human receiver is capable of processing information through only one sensory channel at a time, although the theory postulates a rapid switching mechanism through which the receiver can switch from one channel to another. Accordingly, it is possible to overload the human information processing system by sending messages through multiple sensory channels simultaneously. This condition might result in a loss of information processed by the multi-media CAI interface should be capable of keeping the multiple modalities distinct. This problem is also of concern to the CAI course author who in preparing his graphic and auditory displays must give attention to the information processing limitations of his student.

Another stream of research in psychology has been concerned with the problem of imitation or observational learning. Bandura and Walters[18] have demonstrated that much learning results from the child's observation of the behavior of peers or adult models. Undoubtedly, much classroom learning results from the students' observation and imitation of the behavior of other students and the teacher who serve as effective models. Bandura and Walters have also demonstrated that observational learning occurs as readily from film-mediated models as from live models. These results strongly suggest that an effective instructional environment should provide opportunities for students to observe and imitate the performance of models. To provide adequate opportunities for imitative learning, a CAI interface would have to provide a video tape, closed circuit television, or film projection capability. Provisions for observational learning would seem to be most valuable in science instruction. In learning to use various pieces of scientific apparatus or measuring instruments, the student could watch a video taped demonstration by the teacher and then practice imitating the teacher's behavior. Ideally, such a system should have a playback feature so that students who require more than one demonstration can have the demonstration repeated. Slow motion and stop action would also be of value in breaking down the components of a complex demonstration so that the student can observe the subtleties of the performance.

Another important aspect of the instructional process is that of guidance or prompting of the desired response. The interface must be capable of providing hints or cues to the student when an unusually long response latency indicates that the student is confused, or when the student makes an overt error. Hints may consist of highlighting the correct response in a display by increasing its brightness or size relative to the other words in the display. IBM's *Coursewriter* provides a feedback function which enables an author to provide cues in the form of some of the letters in the correct response. Thus, a student who was unable to name the capital city of New York State could be prompted with the display: A----y. If the student is still unable to produce the correct response after the first prompt, he could be given still more information in the second prompt, such as: A-b-ny. In addition to the guidance provided by prompting, it is also essential that the system provide for the gradual withdrawal of guidance or the "fading of prompts." A prompt should be gradually eliminated over a series of practice trials as the student becomes more certain of the correct response.

Finally, the interface should be capable of providing various forms of positive reinforcement or "rewards" to students as they progress through the course. Reinforcement may often take the form of information to the student concerning his progress through the program. Reinforcement has the effect of sustaining motivation and consequently performance on instructional tasks. Perhaps the most potentially powerful source of reinforcement in CAI is the degree of control which the student can exercise over the course of instruction, the extent to which the student can select his own menu of instruction, and the extent to which the student can query the computer for information. Recent research on problem simulation and economic games programmed for computer presentation (Feurzeig[19] and Wing[20]) suggest that instruction involving conversational interaction and inquiry are highly reinforcing to students. Although these interactive programs place no unusual demands on the stimulus display requirements of the interface, they do extend considerably the requirements for response processing. The interface must be capable of accepting rather lengthy verbal inputs, and the computer must be able to detect key words in a wide variety of verbal responses.

Instructional objectives

Instructional objectives play an important role in determining the interface characteristics required in instruction. It is no great surprise that one of the most successful early CAI projects involved instruction in arithmetic drill (Suppes, Jerman, and Groen[21]). Arithmetic drill provides a relatively delimited class of instructional stimuli and responses. The presentation of arithmetic problems via the interface presents few serious display problems, and the responses are such that little complex response processing is required. As Suppes has demonstrated, elementary arithmetic skills can be easily taught by a relatively simple teletype interface, which pro-

vides an important adjunct to regular classroom instruction. As one begins to include other instructional objectives such as diagnosis, remediation, application, transfer, problem solving skill, attitude formation and change, etc., one finds that the interface facilities must be broadened considerably to provide the enriched variety of experiences needed to accomplish these objectives.

SUMMARY

The primary purpose of this paper was to illustrate the effects of different subjects, different students, different learning objectives, and instructional processes on the functional requirements of the student-subject matter interface. Although there is much diversity in the experiences which an instructional environment must provide to students, these experiences also have much in common. The question arises as to whether a single general purpose instructional interface can be developed to teach a wide variety of different subject areas. Glaser, Ramage, and Lipson[1] have addressed themselves to this question. In general, there seems to be some concensus that the next generation up-to-date operational CAI system will have the following capabilities;

(1) Keyboard for response input
(2) Cathode ray tube display with light pen response capability
(3) Video tape or closed circuit television capability built into the CRT unit
(4) Random access image projector with positive address (may be unnecessary if the system provides video tape capability)
(5) Random access audio communication device with positive address.

A CAI system with the above capabilities would be able to provide many of the educational experiences outlined in Table I.

In analyzing elementary mathematics, reading, and science, Glaser, Ramage, and Lipson[1] have outlined some of the major deficiencies of the general purpose interface. These special educational functions may require special purpose interface devices which will be feasible only in experimental CAI systems in the near future. An operational instructional system may have to provide these experiences through regular classroom instruction or special laboratory experiences.

Major deficiencies of a general purpose student-subject matter interface:

(1) Mathematics—Object manipulation to develop basic number concepts; the manipulation of three dimensional geometric figures, line drawings, bisection of angles, drawing perpendiculars, etc. as in geometry. The "Rand Tablet" which is currently operative on some experimental CAI systems provides the graphic response capability required for teaching courses in mathematics, science, and handwriting. The Rand Tablet allows the student to make line drawings, graphs, and diagrams which are simultaneously displayed on a cathode ray tube screen, and evaluated for accuracy by the computer. The Rand Tablet may be operational in future CAI systems.
(2) Reading—Oral response detection for very young children.
(3) Science—The limitations of the general purpose interface in the area of science instruction depend upon the extent to which actual experiences with scientific phenomena can be simulated. Will student learning of scientific concepts be adequate in simulated laboratory experiences, or will direct experience with the actual phenomena be required? Although a number of investigators are presently engaged in the development of simulated science laboratory programs for CAI, these programs have not as yet been evaluated, thus, we must await an answer to the question of the value of simulated experiences in science. One recent doctoral dissertation completed at Penn State has some bearing on the issue of simulated versus actual experience with scientific phenomena. Brosius[22] compared the learning effects of watching films of the anatomy and dissection of the earthworm, crayfish, perch, and frog in biological science with the experience of having the student perform the actual dissections of the animals. Student achievement was measured in terms of achievement of factual information of the anatomy of the animals, skill in the performance of actual dissections, skill in the manipulation of scissors, scalpel, forceps, and probes, and attitude towards science. The filmed demonstrations were as effective as actual dissection on all measures, except the achievement of factual information in which the film method was actually superior to the method involving actual dissection. This study suggests that filmed demonstrations may be just as effective as direct experience in facilitating learning of some concepts in biological science.

Although present technology may be inadequate to accommodate all instructional applications outlined in this paper, it is expected that improvements in the interface will emerge which will closely approximate the requirements of many educational tasks.

The writer's general position is that the interface should "ideally" provide as wide a latitude of stimulus display and response capabilities as possible to accommodate a variety of instructional problems. However, while we wait for the necessary technology to emerge much valuable work can be accomplished with relatively simple interface devices. As we work with first generation equipment we must be especially alert to its limitations in planning the objectives of instruction. Experiences which cannot be adequately provided via CIA must be provided by other "off-line" methods, such as textbooks, workbooks, educational television, laboratories, teacher demonstrations, and maybe even a lecture or two.

REFERENCES

1 R GLASER W W RAMAGE J I LIPSON
The interface between student and subject matter
Learning Research and Development Center, University of Pittsburgh Pittsburgh Pa pp 177 1964

2 H E MITZEL
Five major barriers to the development of computer-assisted instruction
Experimentation with Computer-Assisted Instruction in Technical Education Edited by H E Mitzel and G L Brandon University Park Computer-Assisted Instruction Laboratory The Pennsylvania State University Semi-Annual Progress Report pp 13-19 December 1966

3 H E MITZEL
The development and presentation of four college courses by computer teleprocessing
University Park Computer-Assisted Instruction Laboratory The Pennsylvania State University pp 94 1966

4 H E MITZEL G L BRANDON
Experimentation with computer-assisted instruction in technical education
University Park Computer-Assisted Instruction Laboratory The Pennsylvania State University Semi-Annual Progress Report pp 88 December 1965

5 H E MITZEL G L BRANDON
Experimentation with computer-assisted instruction in technical education
University Park Computer-Assisted Instruction Laboratory The Pennsylvania State University Semi-Annual Progress Report pp 66 June 1966

6 H E MITZEL G L BRANDON
Experimentation with computer-assisted instruction in technical education
University Park Computer-Assisted Instruction Laboratory The Pennsylvania State University Semi-Annual Progress Report pp 127 December 1966

7 International Business Machines Corporation
Coursewriter manual
Yorktown Heights New York Thomas J Watson Research Center 1966

8 K H WODTKE D A GILMAN
Some comments on the efficiency of the typewriter interface in computer-assisted instruction at the high school and college levels
Experimentation with computer assisted instruction in technical education
Edited by H E Mitzel and G L Brandon University Park Computer-Assisted Instruction Laboratory Semi-Annual Progress Report pp 43-50 June 1966

9 K H WODTKE D A GILMAN T LOGAN
Supplementary data on deficits in instructional time resulting from the typewriter interface
Experimentation with computer assisted instruction in technical education
Edited by H E Mitzel and G L Brandon University Park Computer-Assisted Instruction Laboratory Semi-Annual Progress Report pp 51-52 June 1966

10 R E GRUBB L D SELFRIDGE
Computer tutoring in statistics
Computers and automation
13 20-26 1964

11 J J SCHURDAK
An approach to the use of computers in the instructional process and an evaluation
Research Report RC 1432 Yorktown Heights N Y IBM Watson Research Center pp 38 1965

12 K H WODTKE
Correlations among selected student variables reactions to CAI and CAI performance variables
The development and presentation of four college courses by computer teleprocessing
Edited by H E Mitzel University Park Computer-Assisted Instruction Laboratory pp 71-83 1966

13 J PIAGET
The origins of Intelligence in Children
New York W W Norton and Co pp 419 1963

14 D E BERLYNE
Conflict Arousal and Curiosity
New York McGraw Hill 1960

15 R M W TRAVERS
Research and theory related to audio visual information transmission
Salt Lake City Utah University of Utah Bureau of Educational Research pp 477 1964

16 R L R OVERING R M W TRAVERS
Effect upon transfer of variations in training conditions
Journal of educational psychology
57 179-188 1966

17 A P VAN MONDFRANS R M W TRAVERS
Paired associates learning within and across sense modalities and involving simultaneous and sequential presentations
American Educational Research Journal
2 89-99 1965

18 A BANDURA R H WALTERS
Social learning and personality development
New York Holt Rinehart and Winston pp 329 1963

19 W FEURZEIG
Conversational teaching machine
Datamation
10 380-2 1964

20 R L WING
Computer controlled economics games for the elementary school
Audiovisual instruction
9 681-682 1964

21 P SUPPES M JERMAN G GROEN
Arithmetic drills and review on a computer based teletype
Arithmetic teacher
12 303-309 1966

22 E J BROSIUS
A comparison of two methods of laboratory instruction in tenth grade biology
Unpublished doctoral dissertation The Pennsylvania State University pp 151 1965

Centralized vs. decentralized computer assisted instruction systems

by M. GELMAN
Philco-Ford Corporation
Willow Grove, Pennsylvania

INTRODUCTION

This discussion deals with a qualitative comparison of two configuration alternatives in the design and implementation of a Computer Assisted Instruction (CAI) System. With CAI in its infancy and in a necessarily experimental stage, there is little available data upon which to develop quantitative support for the opinions expressed herein. In this respect, CAI bears a definite kinship with "time-sharing" systems. It can in fact be strongly argued that CAI is essentially a special form of the "time-sharing" system. At this stage of CAI development we are still dealing with "guesstimates" of mass storage requirements for course material, quantity of course material required to reside "on-line" at a given time and tolerable response times in the interaction among student, student terminal, and terminal controllers.

This discussion assumes that as a result of a procurement, CAI systems are to be installed in more than one school and that cost/capability tradeoffs are factors in addition to satisfaction of specification requirements.

Two important advantages accrue to a "centralized system" as opposed to a "decentralized system": these are in the areas of mass storage facilities and in the flexibility and potential afforded by the "naturally" formed communications network.

Assumptions and definitions

The key assumption and definitions for this discussion are now stated.

It is assumed that a specification or requirement calls for the installation of one or more complexes of student terminals; i.e., installation in one or more geographically and/or physically separated schools. Further, if the specification calls for only one initial installation, then it is assumed that there exists a requirement for expansion capability of the system to provide for installation of similar systems in additional schools.

The collection of student terminals and associated equipment contained within a single school is referred to as a *cluster.*

A *centralized* system (Figure 1) is one in which cluster operation is dependent to a lesser or greater degree,

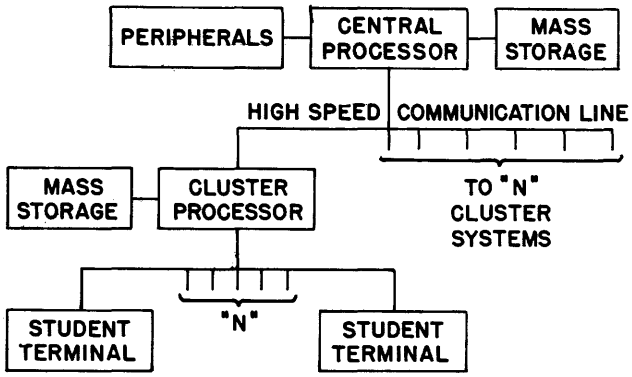


Figure 1 – Centralized CAI system

but nevertheless dependent, upon hardware and software residing in central facilities not part of the cluster, but which are accessible to and shared by more than one cluster. In the centralized system, operation over extended time periods requires the availability of, and participation by, the central facility.

A *decentralized* system (Figure 2) is one in which each cluster is autonomous and independent. All hardware and software necessary for operation is available, and readily accessible from, within the cluster complex. In the decentralized system, operation over extended periods is dependent only upon the availability of resources which are part of the cluster complex.

Succeeding discussion will reference such items as cluster, cluster processor, central, central processor and so forth. Such references mean, in accordance

with the preceding definitions, the resources at a single school, the digital processor which is part of the cluster, etc.

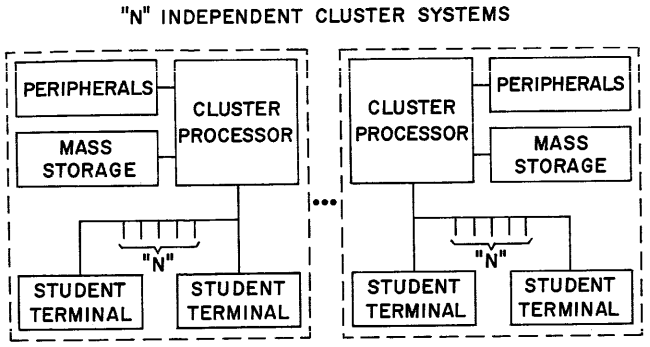


Figure 2 – Decentralized CAI system

Functions

In either configuration the primary functions to be performed include but are not limited to: presentation of instructional material to the student, testing of the students, reaction to student responses, presentation of remedial or enrichment material as a function of student response, construction and/or modification of curriculum by the curriculum generator, student monitoring, collection, correlation, and presentation of pertinent statistical data such as types of response, student response latency time, number of correct answers, and so on, to authorized personnel.

Many of the functions, but especially the presentation of course material, require the use of mass storage facilities. Some of this storage must be rapid, random access in order to satisfy response time requirements as dicated by both student reaction time and the characteristics of CRT-type student terminals.

The storage advantage in the centralized system

The centralized system approach can lead to economies in the allocation of mass storage facilities especially when course material is common to more than one cluster. In a centralized system each cluster requires only that sufficient mass storage capacity be available at the cluster location to maintain its immediate needs of lesson presentation as dictated by student and student-terminal characteristics. The cluster calls for additional material to be transmitted from the central prior to actual need and in accordance with daily schedules prepared in advance. At the central facilities, sufficient storage is implemented through a combination of serial access type storage (tapes), and random access storage (drums, discs), to maintain the current curriculum library. At the cluster, the quantity of storage on-line is sufficient to supply course material to n terminals for one, two, or more hours. The course material may be stored as "lesson segments" of quarter hour, half hour or other duration for each terminal. An analysis leading to the allocation of storage facilities may consider the affects of employing a console usage discipline incorporating combinations of time ordering, console ordering, and student ordering. The designer has a choice varying between two extremes:

(a) Complete freedom of access which permits any student to access any terminal at any time for any lesson in the curricula library. It is apparent that this maximizes both flexibility and local storage requirements.

(b) Restricted access which requires students to access the terminals at a given time for a given lesson. This approach tends to minimize both local storage requirements and flexibility.

It should be noted that in the centralized configuration we are not advocating the complete elimination of mass storage facilities at the cluster; to do so would place the cluster in a position of almost total dependence upon the central facility. The malfunction of the central or of the communication link from central to cluster would then be very shortly followed by a cluster's inability to support required operations. The capacity and type of storage to be located at the cluster becomes a function of central facility reliability and availability as determined by MTFs and MTRs and as bounded by a predetermined minimum capability required at the cluster.

Complete decentralized operation, (independent and autonomous cluster operation), would require storage capacities, at each cluster location, capable of housing the entire curriculum library and data base through some combination of storage media; the cost for such capacity and capability is subject to multiplication in the presence of n clusters. Similar cost multiplication might be incurred in the decentralized system for peripheral devices (printers, punches) which under the centralized system are shared by the clusters.

What is suggested is a deliberate attempt to minimize the cost for storage while at the same time retaining capability for continuing operations at the cluster for a given length of time in the event of central unavailability.

Expansion of system facilities

The advantages of centralized shared storage facilities will become more pronounced as the number of clusters increase; the cost for the storage is now "shared" by a larger number of users. At some point, however, data transfer facilities, communication interfaces and central processor capabilities may become limited and the addition of clusters will require the expansion of central facilities resulting in a tem-

porary increase of the shared cost until more clusters are added. In considering the centralized configuration, items not to be discounted, but not dwelled upon here, are floor space and power consumption advantages for the cluster with the former almost always at a premium in the modern school.

The expansibility of a cluster operation from an "n" to an "n + m" terminal configuration is dependent upon "processor capacity" being expended in the operation of n terminals. If certain background programs (e.g. statistics, student records) are being run concurrently with terminal operation, then transfer of non-essential programs to the central processor allows the cluster processor function to approach pure student terminal operation. Thus, whatever the unused processor capacity is, virtually all of it can be devoted to terminal expansion subject to physical limits of hardware capability, storage, floor space and power.

Some negative factors

It is apparent that this conservation of storage and corresponding cost savings does not come about completely free of "negative" factors. Part of the price for the centralized system and its attendant advantages is an expenditure not encountered with the "free standing" clusters of the decentralized system. There is the matter of recurring monthly charges for the high-speed data lines from the central to each of the clusters. These data lines will be required to operate in the 2400 bps range; they must be leased from a common carrier and they are not inexpensive. Further, the implementation of even a small amount of rapid, random access mass storage at the cluster carries a cost factor which is not linear with capacity: namely the cost of controllers and interface equipment can be inordinate when amortized over small storage capacities. Careful consideration must be given to actual requirements with respect to transfer rates, access times, formats, and response times.

The centralized configuration introduces additional elements of complexity into the computer programs which are an integral part of the system. At the cluster the "executive" routine must now include provision for the communications between cluster and central; part of this routine must address itself to "check functions" to ensure the validity and integrity of transmitted information. The computer-to-computer interface calls for additional hardware. Both hardware and software must be incorporated in a manner which ensures timely transfer of data without degrading the primary function of the system; that is, the interaction between the student and his terminal. The incorporation of the control and bookkeeping routines necessary for the cluster/central interplay must be programmed so that from his position at the terminal, the student, or the instructor, or the curriculum generator believes that the machine is his and his alone.

Availability considerations

While the presence of a central facility permits transfer of considerable functional responsibility (data processing, statistics, curriculum storage, etc.), from cluster to central, care must be taken to ensure that the cluster operation does not become totally dependent upon central availability. Sufficient hardware and software capability must be retained in the cluster in order to establish a satisfactorily high probability that the primary function of providing instruction to students can be carried on for a given minimum time interval subsequent to the loss of central availability. Based on reliability characteristics (MTBF, MTR, etc.), of the central facility and communications links, enough capability should be present at the cluster so that operations can be sustained for a period of time during which a central failure can be detected, isolated and corrected and the central again made available to the cluster before the cluster has run "dry"; that is, before the cluster has run out of useful material. Tradeoffs to be considered here include consideration of hardware redundancy at the central facility versus increased facilities at the cluster. In configuring the system and assigning functions, the designer must continually keep in mind that a failure at the central potentially affects all clusters. Again, the number of clusters becomes an influencing factor; any cost incurred at the cluster level is subject to multiplication.

Other advantages

The communications network

The centralized system configuration presents additional, perhaps more subtle, important potential advantages which are derived from the fact that the presence of a central processor together with high-speed data links to and from the clusters provides the key elements of a real-time communications system among the clusters. The central processor may be viewed as the communications hub or message switch in this configuration.

In this view, we are not thinking of the wholesale exchange of administrative type messages but a rather special communications net which carries "traffic" pertinent to the primary application. The potential thus exists, not present in the decentralized configuration, for a dynamic interaction among the clusters or schools. While the techniques of learning through

intercluster dynamics are not yet defined, the future establishment of group learning sessions with participants in various separated locations communicating with each other will be facilitated by the existence of the communications network within the centralized configuration. As the requirements and techniques for group dynamics through cluster interaction are evolved and understood, implementation through the addition of necessary programs and equipment will be more easily accomplished.

Additional advantages potentially available through the communications network include the following items which I will mention briefly. I believe that others meriting consideration will occur to you.

- The central processor can return to each cluster the statistical results of curricula, presented at other clusters. Comparisons and correlations can be made immediately available to authorized personnel. An extension could include the presentation of identical curricula or tests to geographically separated groups on a "real-time" basis with immediate data collection and distribution of results.
- The curriculum author at one location can "dry run" his material against an author or group at another location. Criticism, suggestions and unanticipated responses can be obtained and acted upon in a rapid and timely manner. This ability to present the material to a "different" audience and to monitor results in "real-time" could materially benefit and improve curriculum construction techniques. Variation in content or order of presentation for a particular course could be tried experimentally at remote locations with on-line monitoring of the activity. Experimental and control groups could be subjected to new situations by varying inputs and testing the efficiency of new approaches. Instructor-to-instructor communication during presentation of material would permit presentation of "on-the-spot" reactions to content.
- The essential equipment elements of a "time-sharing" system are present in the centralized configuration. The availability of appropriate executive, monitoring and scheduling programs would permit the participating clusters to use the system in "off hours" for data processing and problem solving not necessarily associated with the CAI usage during the school day.

A system in development

General description

The Philco-Ford Corporation is currently engineering a "centralized" (by the earlier definition), CAI system for the School District of Philadelphia. At the risk of introducing a new term, the system design may be more properly described as one which features centralized curriculum control with decentralized automatic operations.

A central processor with associated peripheral devices and storage facilities, controls and has available at its location the entire curriculum content which it transmits to each of four geographically separated clusters ("schools"). The transmission of course content, as required, is effected through high-speed data lines (2400 bps) dedicated to each of the clusters and will take place in accordance with scheduled daily requirements of each school. At the cluster, sufficient disc storage is provided to house short-term quantities of curriculum for eight Student-Audio-Visual Instruction (SAVI) terminals; sufficient capability is incorporated to expand each cluster to thirty-two SAVIs. The SAVIs are controlled by a Philco Model 102 stored program digital processor which communicates with the central processor. Operation at the cluster is "independent" of the central processor only in the sense that the cluster processor is completely capable of, and responsible for, driving the SAVI terminals. The latter capability at each cluster is the previously referenced "decentralized automatic operation." Communication with the central is primarily for replenishing the supply of curriculum content for subsequent presentation to the student at the SAVI.

The central processor

The central processor will be used to develop, produce, store and distribute the curricula to the clusters. It can also do problem solving, trends analysis, statistical and arithmetic functions. It communicates over data lines to the cluster processors and will accept on-line inputs from the clusters under program control.

The peripherals which function with the central processor include tape devices, high speed printers, punched card equipment and a keyboard entry device.

The "clusters" (at each of four schools)

The cluster functions are to automatically control the student terminals and to provide inputs to the central processor. Principal equipment elements are the cluster processor and the student terminals (SAVIs):

The cluster processor controls, through stored program, all inputs and outputs generated at the cluster. Inputs are accepted from the central processor and directed to appropriate memory or the terminal. The curricula, including display data, are stored on workable segments in the mass memory and are subse-

quently directed to the SAVI terminals by the processor.

The student terminals (SAVIs) provide integrated input-output capability for the student, produce displays and provide a communications link to the processor through its keyboard. An optional capability provides audio input-output utilizing the terminal's sound system which is also program controlled. The console comprises a cathode ray tube, a light pen, a keyboard device, and expansion capability to include audio.

The software

The programs, designed to meet the needs of computer assisted instruction, include:

Functional software for the central processor:
- Multi-station control programs
- Lesson distribution programs
- Data acquisition programs
- Student status interrogation programs

Functional software for the cluster processors:
- Lesson control programs
- Instructional monitoring programs
- Activity recording programs
- Data transmission programs

Curriculum generation programs:
- Initial curriculum generation programs
- Curriculum modification programs
- Curriculum check-out programs
- Master curriculum file protection programs

Test and diagnostic software:
- For the central processor
- For the cluster processors
- For the SAVI terminals.

CONCLUSION

I have attempted to highlight some advantages which are achievable by employing a centralized CAI configuration. These considerations have been made from a qualitative "narrowview" (many factors remain to be considered), which in simple terms means that at this time I have neither facts nor figures to support the viewpoint. It is anticipated that further investigations, analysis and experience will permit a quantification of the key parameters leading to a model for "tradeoff" evaluation. It is intended that future reports will convey the results of our analysis.

If this presentation serves to motivate some of our colleagues in the field of education to learn more about system design and implementation considerations of Computer Assisted Instruction, then the time has been, in my opinion, well spent. Hopefully, this discussion has suggested that there are many facets to system requirements which should be defined before the specification–RFQ–aware-implementation cycle can properly take place. In spite of the fact that systems engineers and designers always cry that they "need more information" there nevertheless exists a need for improved definition in CAI procurement specifications if industry is to be responsive. Only through better definition can the user make a true "apples-to-apples" comparison of the variety of systems, costs and capabilities proposed. If statements of firm minimum requirements are not detailed, the educators may be in for a painful education; the system may meet the contractual requirements as determined by legal counsel but it may fall far short of the user's intent.

The field is in its infancy and some of the needed improvement will evolve with experience; but frustration and disappointment in the interim can be minimized only through concentrated and deliberate effort. It has been said of industry, with justification, that "their ignorance of education is gargantuan. But they impress me. They'll sit down and learn it."* Similarly, educators have a need to be better informed about computer technology. The two fields have much to offer and teach each other. Only through the communication and understanding of the disciplines involved can educators and industry provide the effective and efficient CAI systems to help meet the demands for quality education in the face of an expanding population.

Government at all levels, the nation's educators and industry are involved in the embryonic phases of a nationally significant joint venture. In a keynote address at the 1966 FJCC, Ulric Neisser suggested the image of the computer expert awakening in the morning, looking into a mirror and smugly thinking, "I have social impact." True or false, through evolution or revolution, there should be no question that CAI will have social impact.

It should be obvious that the great potential afforded by CAI also presents a significant technical challenge and awesome responsibility. It should be equally apparent to all concerned and to our industry in particular that any attention to the "rustle of the new money that is falling like autumn leaves on the educational ground"** must be transcended by an imaginative response to the technical challenge and by a mature fulfillment of the civic responsibility.

*P. Suppes, quoted in *The Computer as a Tutor,* Life Magazine, January 27,1967.

**_The Computer as a Tutor_, Ezra Bowen, Life Magazine, January 27, 1967

Reflections on the design of a CAI operating system

by E. N. ADAMS

IBM Watson Research Center
Yorktown Heights, New York

INTRODUCTION

A time shared general purpose computer programmed to operate in conversational mode may be used as a vehicle for computer aided instruction (CAI). Although at present CAI is a relatively expensive medium the outlook is that as the efficiency of time sharing system increases, the cost per terminal of such a CAI system will decrease and CAI will become practical for more and more educational applications. As a result R&D activity in CAI has increased markedly in the last few years.

Current R&D in CAI is being addressed to a number of important problem areas: system design and terminal capability, programming languages and procedures, pedagogical techniques in relation to various subject matter areas, problems of operational use. Most current CAI work is not especially concerned with cost effectiveness and operating efficiency, since in an R&D phase one is more concerned with feasibility and effectiveness than immediate practicability and efficiency. Nevertheless, we think a discussion of cost effectiveness and operating efficiency is worthwhile as an avenue of insight into the wise use of CAI.

Cost effectiveness of a CAI program is perhaps best approached conceptually in terms of the strategy of the instructional program and how well it makes use of both human time and system capabilities. In particular, since an information system is a high-cost medium one expects that where good cost effectiveness is achieved it will be through a well conceived exploitation of the system capabilities which are unique to a computer based system:

- Means of input and display which permit flexible man-machine communication,
- Capability to process and respond to messages written in natural language,
- Capability to rapidly evaluate complex mathematical functions,
- Capability to record, analyze, and summarize student performance data, and
- Capability to administer programs of instruction in which flow of control is contingent on a variety of program parameters and indices of performance.

We cannot discuss operational efficiency of a CAI system in such general terms, but instead must consider how the physical configuration and the programming system conform to the pattern of instructional use of terminals and processor: an optimal system design for a program which primarily exploits the display capability would be quite different from that for a program which primarily exploits the processing power.

We have looked at a number of CAI programs prepared for use with the IBM 1440, and IBM 7010-1440, and the IBM 1500 to see what generalizations about system design they might suggest. These programs were all designed to be used with a terminal which was either (a) a typewriter, (b) a CRT-keyboard station with light pen input coordinate sensing capability, or (c) either (a) or (b) supplemented by an A/V device which could display prestored output in the form of still pictures of audio records. (The authors of these programs did not have available, for example, a "Sketchpad-like" capability for graphics manipulation, a large simulator, or any other technical feature using truly large processing capability). We think that as a group these programs may be representative of a broad class of CAI programs which can be executed on a system of modest cost.

The instructional programs we have examined exhibit several basically different structures:

- A "scrambled book" structure, in which the

program consists of small presentation-and-test units ordered in a defined sequence, but with an occasional "test and branch" unit to transfer control from one sequence to another as appropriate on the basis of student achievement.

- A "drill and practice" structure, in which the program consists of modules, each module containing a pool of similar exercises. Items within a given module need not be presented in a prescribed order but, may be drawn from the pool in a random way. Thus a student does not "go through" a module, but "practices within it" until he attains to some statistical criterion of mastery. Within such a module the program is conveniently structured into a "text block" and a "control block." Drill and practice programs present a wide spectrum of sysm requirements in the areas of terminal characteristics, system response rate, processor load.
- A "diagnosis and decision" structure, in which the student is required to carry out a complex task consisting of a number of subtasks that are logically related and may have to be carried out either wholly or partly in a prescribed order. Such a program typically does not consist of a sequence through which the student must proceed, but of a network of program blocks and of logical trees. Examples of such programs are chemical analysis or medical diagnosis.
- A "simulated laboratory" structure, in which the student "expreiments" with a mathematical model of a process or processes. Such a program may contain simulator blocks, request handling blocks, and tutorial blocks. It may make use of auxiliary equipment used either on or off line, and the student may exercise considerable initiative as to what part of the program has control at any moment. Business games represent a widely familiar example of simulation program.

The quantitative technical requirements of these programs vary widely, so we are limited to qualitative conclusions in regard to what the system program ought to do. Combining these with some frequently expressed requirements of program authors and student users, we compile the following list (non-exhaustive) of system features which are important for design of the operating system.

- *System response* usually should be "very fast," in the sense that the student should be working much more than he is waiting. Some users specify a system response time of a small fraction of a second; in many cases we think it may suffice to have the machine reply to a student in a tenth of the time it took him to frame a message.
- *Source language* must provide a number of functional requirements, some of which will be listed below. The most difficult questions regarding source language involve how "easy" it is for various classes of users to avail themselves of various features.
- For the *source language compiler*: capability to compile programs entered either "off line" or "on line," very fast compile service where small emendations are made to a program.
- *Transaction recording*: capability to mark and specify content of transaction records which may include time data, contents of storage, text of messages; simple procedures to request automatic retrieval, analysis, and summary of data in records.
- *Recovery from malfunction:* in addition to features generally desired on other time shared systems, special provisions to recover recent contents of working storage after system malfunction; also, especially in system configurations having remote terminals, a number of features to inform users of system or terminal status in event of malfunction.

Of the performance requirements referred to above, one which is troublesome to realize quite generally is that of fast response times. CAI programs are commonly very long: the stored program to process a single student message may consist of hundreds to thousands of instructions of machine code. Moreover because each of the many users of a program will need his own working storage, a single tutorial program may require a very large amount of machine storage, usually peripheral storage. Thus a central problem in a design of the operating system is how to provide adequate working storage for an instructional program and how to quickly get into core all the necessary program and information from peripheral storage at program execution time.

In the disc oriented systems we have worked with, the preferred location of all programs not being currently executed is on disc. The core memory is partitioned into an area for the monitor program and areas for programs to be executed under the monitor. A "course" program to supervise a given student is normally brought into core only when his terminal indicates to the system that it has completed sending in a message; at that time an appropriate part of the course is copied into a core area allotted to his program. to his program.

Core allocation tradeoffs to speed up terminal service are involved at several levels in our system designs. These tradeoffs are primarily concerned with making best use of available core storage by a proper

allocation:

- Among monitor, interpreter, and user programs.
- For a given user, among course program, working storage, and routines needed to execute "functions."
- Among competing user programs.

To facilitate making the tradeoffs between monitor, interpreter, and user programs we made a distinction between two kinds of routines which constitute our source language: common routines, (which are "operation codes"), and infrequently used or very lengthy routines, (which are "functions"). The two kinds of routine are compiled and executed in different manners. The "operation codes" are either compiled directly into machine language or translated at execute time by an interpreter which is always resident in core: in either case, the machine code necessary for execution is all in core at execution time. The "functions" on the other hand are left untranslated in the compiled program and are executed interpretively by special routines which are fetched at execution time, either from disc or, in special circumstances, from elsewhere in core. By making a routine a "function" rather than an "operation code" we trade off the extra delay incurred when it is necessary to fetch the routine from disc against the core space necessary to store and interpret it.

In allocating core storage for a given user the "page" size of program is the natural parameter. In determining the page size one should consider not only the total number of instructions in the user program but also the number of these which must actually be executed in processing a typical student response. Thus it might be that to deal with the expected range of replies to a particular question the program would have to contain 1000 instructions, and yet that any of the replies most commonly given by students would be completely processed by the first 100 of these instructions. In all these common cases then, the remaining 900 instructions are not required in core. From such examples we are led to consider choosing the page size as small as possible, so long as the most commonly used answer processing strategies can be brought into core on one or two pages, even though an occasional slow system response will be experienced by the terminal user who has made an uncommon answer.

A given user program will suffer incremental delays in execution (which contribute to the response delay) because of every reference to disc, whether for contents of previous working storage areas, for routines to interpret "functions," or for more program to execute, so in principle one should tradeoff all three of these delays in making core allocation. However, in our system designs we have thus far always taken the approach of providing only a small amount of storage (of the order of 1000 bytes) for bit switches, numbers, buffers, etc., but making it all available in core at execution time. Thus our chief tradeoffs involved the "page size" of program brought in, against the average probability that the program to interpret a desired function would be in core, and both of these against the number of programs using core at one time.

It may seem obvious that if the processor must commonly make several references to disc in the course of processing a single student message there would be definite advantage in processing several terminal programs simultaneously, since during the many milliseconds the disc arm is in motion seeking more program to service one terminal, the processor could be processing program for another terminal. Clearly such multiprogramming would permit the system to avoid the situation that all users experience extremely long response delays whenever a single "problem" user requires an extremely long time to have his terminal serviced. However, when core is divided among many programs one pays a price for frequent interruption, competition of programs for disc arms, etc., and as the available core is divided among more and more user programs, and as the average amount of core available to each one goes down correspondingly, a point is reached beyond which the average user will get *slower* terminal service. This point is reached roughly speaking when every user requires on average multiple fetches of program from disc to service a single request. Thus the optimum number of programs to share core is a function of the amount of core to be shared; if there is very little core available altogether, multiprogramming may actually result in *slower* terminal service on average.

The programs we have seen represent such a great variability of structure, that no single storage allocation could approach being optimal for all of them at once; the best allocation would clearly change from user to user, hence, from hour to hour as the courses loaded on the system change. It would be very desirable, therefore, for the monitor to provide means for the system operator to vary at will (a) the maximum number of user programs which will be simultaneously serviced, (b) the number (and if desired the identity) of functions which will be resident in core, and (c) the "page size" of the executable program.

We may indicate the magnitudes involved here by means of an example. Our 7010 programs require that there be stored somewhere from 900 to 30,000

characters of compiled machine code to process the answers to a single question. This code is on disc and is brought in a page at a time when an answer is being processed, our page being 900 characters at present. Many programs use several functions for each 900 characters of compiled code, each 7010 function itself consisting of 900 to 3,600 characters of code. Clearly in executing programs which use so many functions, more function code comes into core than program code. In seeking how to balance time spent in fetching functions with time spent in fetching new programs we concluded that for many of our programs on the 7010 10-20,000 characters of core for "recently used functions" would suffice to adequately limit the disc arm movements involved in the necessary fetches. The page size for correspondingly good program access rate would range between 1,800 and 4,500 characters. With the amount of core space we have for working storage we would have room for 2 to 4 user programs after optimizing the size of the page and the core block reserved for functions. We are still making system measurements to improve our understanding of these tradeoffs.

We have not time on this program to explore any of the complex questions involved in source language specifications. We will merely enumerate some of the functional capabilities the programming system must provide:

a. A convenient means of programming basic "question-and-answer" sequences is very important for all programs. A "question-answer" sequence commonly begins with a question, (i.e., a request for student input) which is followed by a series of program blocks, each consisting of a single strategy for reacting to the student's answer. An individual strategy typically begins with a *test* of the student answer and a *branch* on the outcome; if the test "succeeds" an associated sequence of statements is executed; if the test "fails" control passes immediately to the point in the program where the next strategy begins.
b. Capability to edit, search, transform, and extract from student messages in a variety of ways. Such capabilities are essential so that the student can communicate in an other than monosyllabic way and in regard to a variety of tasks without cumbersome formatting and coding procedures.
c. Working storage which can be addressed by the instructional programmer in which he can store various kinds of information: decimal numbers, clock readings to tenths (preferably hundredths) of a second, address locations, alphanumeric messages, logical conditions expressing "states" which have occurred in execution of the program.
d. Means for the instructional program to test and branch on a variety of conditions involving stored information: bit switch values, numerical inequalities, alphabetical order.
e. Capability for the instructional program to do arithmetical operations on stored data.
f. Capability to execute as part of a source language program special routines written in lower level language and to add new routines to the source language at will.
g. Capability to efficiently search, mark, and modify complex text structures.

Our system users have had much experience with the entry of new program material, both on-line or by cards punched off-line. A number of users find that for the entry of original program material card entry is preferable. However, there is consensus that the combination of on-line entry and immediate instruction-by-instruction compiling (as in the IBM 1440 and 1500 Coursewriter systems) is very valuable when an author is revising a program, since a large number of minor program corrections must eventually be made, and one wishes to shorten the turn around time for proof and revision. For other problems of program preparation on-line compiling is not a panacea.

The process of on-line revision of a program commonly results in its being located in an undesirable pattern on disc (one for which response times will be long), and leaves the author without a simple documentation of the program. Thus it seems that an author must normally end up making complete recompiles of a program after significant revisions are complete and before normal use.

Logging and analysis of transaction data is an especially important function of a CAI system. There are several different kinds of data request which seem to be important, corresponding to different functions for which a CAI system may be used:

- Program requests for purposes of program control, e.g., to affect a student's progress through the next units of program.
- Teacher requests concerned with class supervision; these may involve brief summaries of individual performance statistics and a report of the distribution of the class members within the program at a definite point of time.
- Teacher requests concerned with student counseling or evaluation; these may involve a detailed record of performance statistics for an individual over a period of time.

- Author requests aimed at item evaluation and improvement; these will involve actual texts of student messages and distributional data concerning the processing of the messages by various blocks of program.
- Educational research requests; these may involve any of various kinds of data, and may be organized by item or by students, possibly with reports of correlations and summaries of correlation between particular data sets.

For purposes of designing the logging and analysis system the most important difference between the various kinds of data requests is not in the nature of the data needed, but in the time at which requests are made and serviced in relation to the time that the data are taken. Roughly speaking there are three kinds of requests:

- Requests to be processed as data is taken, such as those in current "course registers."
- Requests which must be processed very quickly, such as weekly status reports.
- Requests which may be processed on a convenience basis as the necessary files are made up, such as large data compilations for authors or researchers.

Two kinds of files are required to service these requests: files which are always current in the system and files which are made up from transaction log tapes. Data summary request procedures will be different for the two kinds of files: the procedure for querying current files is via on-line author service programs or utilities, and request must be serviced when the CAI system is operating; the procedure for querying tape files is an "off-line" request probably via cards to a program which sorts and searches tape files.

The file preparation and sort programs to make these data summaries convenient to get will be fairly elaborate. For greatest convenience of use the programs should provide for accepting requests having a fairly flexible request format and should produce data summaries which also provide for a range of formats. Unfortunately, our understanding of the design of these programs is still quite limited, and our present programs leave much to be desired.

The problems of avoiding and recovering from malfunctions are big ones for all time shared systems, so it would be redundant to discuss most of them here. Probably the single most crucial recovery problem for CAI is that of reconstituting a memory area which a program was using for working storage at the time of malfunction. The situation of the student whose "terminal record area" is lost is not entirely analogous to that of the user who loses his work area while doing computation. This latter person normally has many of the results he had previously obtained in the form of a printout at his terminal, so need only repeat certain portions of the work. The student, whose records are lost, by contrast, has no knowledge of or access to the memory areas which control his program, so has no choice but to repeat in detail a stale conversation.

We consider that the aversive affect of having to repeat a long conversation is very great, so we feel it is essential to keep exact copies of the vital terminal status records somewhere in the system and to update them after each interchange between student and machine. Then when a system malfunction does occur it will be possible to restart the program at a point in time no more than a minute or so before the malfunction.

While "cleanness" of recovery is the most crucial reliability requirement for CAI, rapidity of recovery is also very important. It is worthwhile spending considerable programming effort to arrange that minor malfunctions can be detected and corrected without operator intervention since then recovery can commonly take place in seconds and only one or a few users would ordinarily be aware that a malfunction occurred. Most malfunctions requiring operator intervention can still be recovered from in about one minute without serious inconvenience to users, if operator procedures are good and the operators have suitable utilities available for the purpose. However, malfunctions which require all users to reinitiate their programs are relatively much more annoying to the users, who may lose four or five minutes on account of the malfunction. We feel that a few of these per day, perhaps as many as one each hour, can be tolerated by the students if recovery is "clean."

A certain number of malfunctions are going to occur which cannot be so quickly recovered from, especially in a project where there is experimental hardware or an active program to develop new features in the source language. In our 7010 operation we have experienced "big" troubles causing time losses of the order of an hour or more in a week about once in two months. Since experience indicates that addition of new "functions" to the source language will be a continuing need we should assume such troubles will occur on any system.

It is most desirable that whenever any observable malfunction occurs, the system gives the user information as to whether the delay will be a big one as soon as feasible. If, as is more than likely the case, the user is scheduled into a definite time slot of a half-hour or an hour, he would often prefer to just leave

if the delay is going to be an extended one. For a user at a remote station it is desirable to have a status query capability in the system, so that the user can easily learn the status of the system. On the rare but big malfunction which takes more than a few minutes for recovery, an "answering service" for status queries from terminals is also valuable.

In conclusion I repeat our views of requirements have been developed in a certain framework of computer size, terminal capability, student population, and role of the computer in instruction; we are aware of a wide range of potential applications of CAI to which our experience is only partly applicable.

Nature and detection of errors in production data collection*

by WILLIAM A. SMITH, JR.
Lehigh University
Bethlehem, Pennsylvania

INTRODUCTION

Improvements in data processing equipment, particularly large volume memory storage devices, have heightened interest in the advantages of the single recording of data at its point of-origin. Automated capturing of production data, its subsequent processing into information, and interrogation and display of that information have been recognized as a means of circumventing the restrictions of using historical data and of satisfying the feedback requirements to allow control of the dynamic, evolutionary production system. Historical data, generally unsatisfactory for control, can be characterized as:

(1) Inaccessible from delays in processing,
(2) Distorted from processing for accounting purposes,
(3) Inaccurate because of error in recording (estimated at ±5%),[1]
(4) Incomplete for some important variables,
(5) Obtained under normal or conservative conditions,
(6) Not representative of some short or long cycles in a process, and
(7) Invalid because of subsequent changes.

The automatic recording process must provide for relatively low speed transfer of data from multiple input stations to a central processing unit in machine usable forms. In general, the capturing or gathering functions have become classified as:[2]

Data collection—mostly digital data from human sources; control at remote stations; associated with production control.

Data acquisition—mostly analog data from varied electromechanical sources; control at central station; associated with process control.

Data collection is appropriate when judgment or visual observation is necessary or when mechanized methods are impractical or too costly. In general, the human reports unusual events or vital normal events at his input station. Most equipment available will allow a variety of input media such as:

(1) Time and date automatically from a clock.
(2) Fixed identification codes automatically from input station.
(3) Semi-fixed or pre-assigned data from plastic badges, punch cards, etc.
(4) Variable data from levers, dials, keyboards, etc.

The advantages of automated source data collection are normally considered to be increased accuracy in recording facts, more complete description of events, and reduced time in making data available for further processing.[3] Emphasis is placed upon capturing production data with a minimum lag between occurrence of the event and its report into the system. This process bypasses cinventional data processing and supervisory verification[4], assuming that the advantages of immediate recording at point of action will nullify this loss. In general, automation of the process increases computer memory requirements, transfers error checking from manual to computer methods, and creates additional need to handle multiple inputs. The hardware requirements are complicated by needs to transmit large volumes of data, to accept and adjust to unusual variations, and to provide feedback to the person generating the entry.

For this study, the data collected in three different production situations, one assembly shop and two job shops, were analyzed and the characteristics and quantity of errors were determined. The recording

*This paper is adapted from a thesis accepted by the faculty of the Graduate Division of the School of Engineering and Science of New York University in partial fulfillment of the requirements for the degree of Doctor of Engineering Science.

practices and terminal equipment varied among these shops. A controlled experiment was also conducted on thirty subjects recording messages to determine the generality of accuracy found in the production shops.

Characteristics of errors

Beyond the problem of achieving accurate, reliable input, however, is the issue of the effect of existing data inaccuracy upon the information system. Frequency of mistakes in digits entered into the system is important in evaluating and comparing similar collection devices, but digits presented to the information system can, and usually do, have different significance. For instance, a single digit mistake in a lot number or product identification code causes more consternation than a mistake in the number of defects of one kind found at inspection. A mistake in the least significant digit of a reported quantity causes less message reconstruction than an improperly entered code to identify the type of transaction. Thus, it is insufficient to determine the overall percentage of error entering the system. Mistakes must be categorized by their recognizable traits and by their relative cost to the information system.

The errors found in data collected in field studies fell into three major categories which were used in analysis and classification of data accuracy.

(1) *Message Format*—Entries with wrong format that can be detected and screened from system input (wrong message length, illegal characters, equipment malfunction).

(2) *Message Content*—Entries that have correct form, but can be detected as logically inconsistent (shop status contradictions, unusual quantities, corrections to good entries, wrong machine or operator designation).

(3) *Event Description*—Entries that have correct form and are logically processable, but prove inconsistent after subsequent entries (omitted entries, extra entries, wrong sequence, late or early entries, failure to correct detected mistakes).

Errors listed in the first category above are relatively inexpensive because they can be readily identified and transmission can be prevented or the need for correction can be signalled to the operator before he leaves the input device. When these mistakes occur in manual information systems, they are usually corrected by pencil and eraser. Attempts to record faulty format messages are easily screened. However, 100% inspection of entries is required in order to strip these mistakes from data entering the processing system. Entries that appear to be acceptable must be subjected to further editing to test the logic, or reasonableness, of the content of the message. Detecting inaccurate or inconsistent messages in this fashion also requires analysis of other entries. The costs of this checking are greater because each message must be analyzed and compared to shop status information and to expected ranges of values. This normally requires computer system time and compounds the queueing of entries by requiring more processing time for each entry. The degree of sophistication ological tests can be arbitrarily selected, but improper messages that are not detected and corrected at this stage necessitate special audits or remain undetected. The latter error entry is the most expensive to isolate because review of previous entries or holding an entry in suspension for later confirmation is usually required. This kind of mistake not only requires additional processing time, but it can also cause false indications of illogical entries among subsequent messages processed. Memory requirements are increased, depending upon the volume of such mistakes and the length of time messages must be held. Failure to detect mistakes or to resolve in compatibilities cause bias in recorded production status. Entries that remain uncorrected or undetected after normal processing of data are defined as residual errors.

Discussion

The development of categories to characterize data collection errors occurring in actual factory conditions provided a consistent definition of error for a variety of production situations. Format mistakes, which were most prevalent, contributed substantially to the input load on the system accepting the entries, but were predominantly corrected. Content errors in messages included miscounts and misidentification as well as manipulative mistakes in entering intended digits. Faulty and omitted description of events, although few in number, proved difficult to detect and correct, contributing about half of the residual errors and increasing processing time per entry. Many content and event description mistakes were not detected during normal processing and, even when detected, the majority was not properly corrected (See Figure 1).

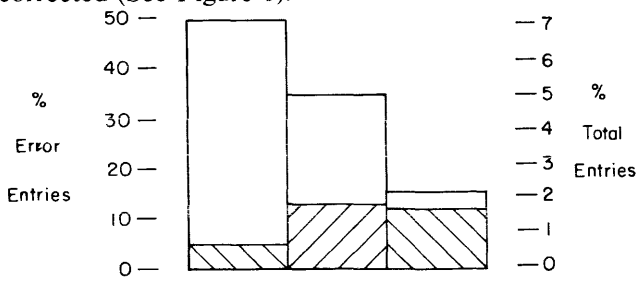


Figure 1—Field study errors by category

The large proportion of errors which were detected and corrected at time of entry into the device suggests the desirability of immediate error checking while the worker is making an entry. A substantial reduction in system load can be achieved by screening detected format errors, including station calls with no message, from further processing at the time of entry. Although errors recognized during processing were a lesser proportion of entries, there was a tendency for more than half of them to escape detection and to be residual errors in the processed information. The further an error penetrated into the information system the more difficult it became to detect and correct. More attention must be directed to effective procedures for analyzing and correcting errors that escape detection by the worker or data collection device at the point of entry (See Figure 2).

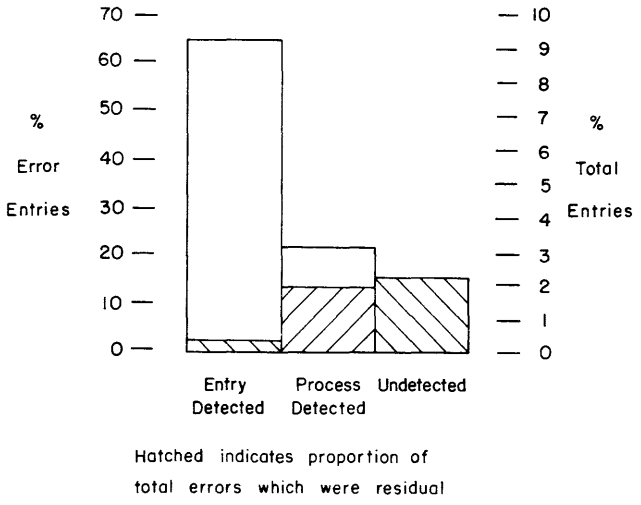


Figure 2 – Field study errors by source of detection

The residual error rate for those mistakes persisting after detection and correction procedures, was consistent at slightly more than 4% for production and inspection workers in situations intensively studied.[5] This was true although the distribution of total errors showed statistically significant differences between shifts and between different production situations. In the only test where the mean residual errors did show a significant difference, recording practices on two shifts were pronouncedly contrasted. Thus, it appears that the proportion of persistently wrong entries, consisting of residual uncorrected and undetected errors, is not dependent upon the differences between production environments for system operating in a steady state.

The high efficiency in correcting errors detected at the time a message was recorded found in field studies was sufficiently universal that it was reproduced and confirmed under laboratory conditions.

Table I
Summary of Experiment Residual Errors

	Untimed	*Timed*	*Total*
A. Entries	5508	5474	10,982
B. Errors	192	217	409
Entry detected	109	86	195
Uncorrected	1	11	12
Undetected	83	131	214
Residual	84	142	226
Error Rate (%A)	1.5%	2.6%	2.1%

Also, the relative contribution of format and content mistakes to residual error was similar to that shown in Figure 1 for the field studies. Because the experimental task involved copying rather than recording events, the opportunity for each category differed from that in field studies but did not alter the residual results except in the event category.

Table II
Contribution of Kinds of Errors to Residual

	Entries	*Total Errors*	*Residual Errors*	*Error Rate*
Experiment:	10,982	409	226	2.1%
Format		118	96	0.9%
Content		288	127	1.2%
Event		3	3	Negligible

The laboratory experiment allowed variation of message length and time restraints in a fashion that could not be achieved in the field studies. Manual entry of a 3 digit message was clearly preferable to manual entry of either 6 or 10 digit messages in considering both the load imposed on an information system by the total errors encountered and the inaccuracy caused by residual errors. Similarly, 19 digit manual messages resulted in insignificantly more total and residual errors than 6 digit messages did. Pressure of time in recording had a statistically significant effect upon residual errors but did not alter the total number of errors committed.

As in field conditions, about half of all the mistakes in observed manual messages under laboratory conditions were caused by single digit substitution. Omission of a digit accounted for another twenty per cent. Transposition mistakes at 7% were more frequently encountered in the laboratory experiment, but they were a less important contributor to inaccurate data recording then generally assumed.[6]

The variability in errors demonstrated by individual workers among different periods of one week in field studies was sufficient to produce standard deviations almost as large as the mean error rate. This masked the possible difference in performance among individuals and indicated sensitivity of data accuracy to subtle influences within a production environment. Temperature and humidity conditions, supervisory emphasis, personnel moves, changes of procedures, and alteration to equipment can be expected to affect

accuracy of data collection. The differences in total error distribution between production shifts further confirms that supervision, or motivation, can have material effect on errors within an operating environment. The differences in kinds of errors between job shop and assembly line operations were not surprising after discerning shift effects within each situation. This investigation was unable to discern preference among situations with production workers recording at their own work stations, machine setters recording for groups of workers on like machines, and clerks recording at a central station. The argument for clerical workers has generally been based on familiarity with the recording process and resulting fewer mistakes in format or procedure. Since the study shows these mistakes to be a lesser contribution to residual error, it would appear that the most accurate data would be generated by persons directly responsible for and with a vested interest in accomplishing and reporting production events rather than those performing primarily data collection tasks.

REFERENCES

1 R G CARLETON O W HILL P V WEBB
Applying a data acquisition-computing system
ISA Journal 10 7 57-60 1963

2 J A PEARCE
Data collection systems their application and design
Journal of the British Institution of Radio Engineers
6 489-496 1962

3 N P EDWARDS
One the evaluation of the cost-effectiveness of command and control systems
AFIPS conference proceedings 25
Spartan Books 211-8 Washington DC 1964

4 F J MINOR S L REVESMAN
Evaluation of input devices for a data setting task
Journal of Apped Psychology 46 5 332-336 1962

5 E T KLEMMER G R LOCKHEAD
Production and errors in two keying tasks: A field study
Journal of Applied Psychology 46 6 401-8 1962

6 C CARLSON
Predicting clerical error
Datamation 9 2 34-6 1963

Serving the needs of the information retrieval user

by C. ALLEN MERRITT
International Business Machines
Yorktown Heights, New York

INTRODUCTION

During the last few years, there have been a large number of reports, papers and articles written on a variety of information retrieval systems. I believe that too much has been written on the needs of information retrieval systems, and not enough either researched or written on the needs of the user. Webster's Collegiate Dictionary states that "need" is a "necessary duty; obligation; or a condition requiring supply or relief." This clearly defines what we mean by the "needs" of the information retrieval user. Despite all the advances in the field of information retrieval, we may inadvertently be building limitations into many of our information systems, simply because we are not fully considering the user, his needs, his reactions and his interactions with information retrieval.

Within the IBM Corporation today, as with many other large organizations, the individual scientist or technical/professional person finds himself a part of large and complex technological endeavors. These endeavors reflect the rapid industrialization of science and the scientist. In this new environment, the scientist's information needs cannot be dismissed as a uniquely personal concern. Increasingly, they have become viewed as a responsibility his employer should share, most particularly with information services that conserve the ever more sought talents and ever more costly manhours of available technical manpower. This change in organizational philosophy has led to the appearance of the modern information retrieval system. We know that the scientist and the engineer are accepting this new philosophy, that is, the judgment of another in providing subject matter through a storage and retrieval mechanization that he does not ever expect to operate personally.

We all know that today the amount of information being generated and accumulated is growing at an alarming rate, and this is bound to continue. Specialization in all areas of human endeavor has increased, while at the same time, the former conventional boundaries between disciplines in science disciplines in science and technology have vanished. Our scientists and engineers need to have access to knowledge related to their particular interests, no matter where it might exist. In this paper, I will explain the establishment of the IBM Technical Information Retrieval Center as a centralized computer information retrieval and dissemination system which services the scientific and technical community of the IBM Corporation. I will emphasize the direct correlation of this organization to the needs of the user; in areas of dissemination, retrospective searching and micro-processing.

Known as ITIRC, our system has two unique points differing from most other information retrieval systems. First, we use a normal text information retrieval technique for searching; that is, questions are expressed in terms of English words, phrases or sentences with logical ANDs, ORs, NOTs, adjacent words, imperatives and a minimum number of matches required to search the normal text. Second, we must really know our user—the engineer-scientist. Knowing our user means that we must not only work closely with him in the formulation of a personal profile, accepting any search requests that he desires to have performed, but we must evaluate each response that he returns to us as a result of material that we have forwarded to him. Our goal is to supply the right information to the right person in the shortest possible time, and at the least possible cost. It is through this type of endeavor that the engineer-scientist can best combat the real problem of technical obsolescence. Our users know that they are living in a fast-moving environment, for today's research idea can be tomor-

row's hardware. They know that information retrieval systems can help them, but not only must they be educated to the values of such systems, but the system itself must establish a rapport with them, so that they, in turn, recognize that they have a certain reponsibility to an information retrieval system. Reflection on the participation of a user in an information retrieval system brings us to the analogy between information retrieval and education. An information retrieval system, like education, deals with communication of ideas and messages. It is imperative that a user be educated to the proper utilization of an information retrieval system. Likewise, the information retrieval system must be educated to the user and the users' needs.

The IBM Technical Information Retrieval Center, henceforth referred to as ITIRC, was established to serve the IBM scientists and engineers throughout the worldwide corporation with a distinct philosophy and approach toward serving the user needs. ITIRC was established in late 1964, after an intensive corporate management study which brought together from three separate IR activities within IBM the best in manpower, experience and systems knowledge available. At that time, a decision was made to use a normal text IR technique developed within IBM, and to establish this newly formed Center on Corporate staff, in order that it might serve the needs of the entire Corporation. The size of the Corporation, the diversification of geographical locations throughout the world, and the vast range of technical interests in research development, manufacturing, and sales provided an unsurpassed challenge to this newly formed group. Immediately upon formation of ITIRC, the decision was made that there would be four basic services established to fulfill the needs of the user.

These were: retrieval, dissemination, announcement and microprocessing and hard copy. The data base for this expanded information retrieval operation included IBM technical reports—those reports prepared by the engineers and scientists as internal documents, IBM invention disclosures—novel ideas submitted as inventions to solve specific engineering problems, IBM released publications—papers or reports presented or published outside the IBM Corporation, non-IBM reports—here selected reports from the total scientific and technical community that are determined to be of general interest to the IBM Corporation are selected as input, and the IBM Research and Engineering project files—the official reporting medium for all research and development activities within the Corporation.

The retrieval or retrospective searching activity is one of the most important functions of ITIRC. This is where a complete search of all current and historical files may be made tailored to a requester's needs. At the present time, this file contains well over 100,000 document abstracts. Search questions arrive daily from users throughout the world, as well as requests from the many local IBM libraries. Upon receipt, these search requests are directed to an IR specialist, who analyzes them and formulates one or more machine questions. Once submitted to the computer, the IR specialist can expect the results, or the answer output of abstracts back within twenty-four hours. He reviews these and sends them directly to the user. The user may find that he is satisfied with the answers, or he may request a deeper search, or he may ask for another search on related topics. At the time he receives his printout of answer abstracts, he receives with it a card on which he can indicate his reaction to that particular question, thus giving him an opportunity to fully express his reaction to his retrospective search answer. During 1966, we processed 3,943 search requests with an average relevance of 85 to 90 per cent.

The most personalized user service that we have in ITIRC is our dissemination program called CIS (Current Information Selection). In this program, the individual engineer-scientist submits a textual description of his information needs and job-related responsibilities. There are three information retrieval specialists or profile experts who are extremely familiar with the sophisticated machine searching techniques that are used, as well as with the total specifications, and thus are able to construct a profile for entry into the system. During the construction of this profile, the IR specialist may, on several occasions, talk directly with the individual user to better formulate his profile needs, or where telephone communication is not convenient, the specialist will correspond with the individual user to ferret out his more specific interests.

I cannot overemphasize that the success of our Current Information Selection program is due primarily to the three individuals who keep in constant contact with the more than 1,700 profiled domestic CIS users throughout the Corporation. Once the profile is operating, it is compared several times a month with all of the current documents being processed in one of four basic data inputs—IBM documents, non-IBM documents, IBM invention disclosures, and selected non-IBM journals and trade publications. Whenever the text of a document abstract matches the profile, a printed abstract is automatically generated and sent to the user. Accompanying each printed abstract is a response card.

This response card enables the user to communicate directly to ITIRC his reaction to that individual document notification. He may indicate that it was relevant to his needs. He may desire a copy of the document, or he may state that the abstract was not of interest.

Our third basic source of information to the user is that of microfilm and hard copy. To all of us who have been exposed to information retrieval in its broader sense, it is a recognized fact that to tell an individual of the existence of a document, and not be able to provide the document when he needs it, is a meaningless type of information retrieval. We must provide the document to the user, either in its original hard copy form or in a suitable microform. Currently, we are putting on 16-mm microfilm all IBM reports, selected invention disclosures and as much of the external non-IBM material as a copyright and distribution restrictions will allow. All IBM library locations have complete files of microfilm going back for several years, and covering more than 35,000 documents. This microfilm is available in 16-mm cartridge or reel form. Thus, to satisfy the user needs, we can provide hard copy to him either through his local library's microfilm, or by his returning to us the response card denoting that he would like a copy of a document. We, in turn, forward to him within 48 hours a copy of the requested document. Where there is a delay in obtaining a copy of the document from an outside source, the user is so notified by correspondence or telephone call.

The library is the natural contact point for the user in viewing and securing his documents, and as much as possible the hard copy needs are serviced by these local IBM librarians. It is obvious by now that ITIRC is essentially a back-up to the IBM libraries throughout the IBM Corporation. We are not only a service to the users, but just as important, we are a service to the libraries.

The final source of information to our user that I wish to discuss, is that of announcement. To supplement our Current Information Selection program, we prepare and publish a series of announcement bulletins monthly. These bulletins cover all current monthly document input processed in the four major data bases previously described. These bulletins are available at all IBM library locations and report centers, as well as publishing groups and some selected individuals who do not have access to an IBM library. These bulletins contain abstracts of documents in accession order; this includes titles, authors, source codes, detailed abstracts and descriptive subject terms assigned by the IR specialist used in the formation of the subject index found in the back of each bulletin. At the beginning of each bulletin, there is a category index with twenty-three broad category terms and easy cross reference by page number back to the abstract. The subject index is an alphabetical listing, including title, accession number and page reference. These indexes and abstracts may direct the reader to his local microfilm station where the complete text of the document is available for viewing. Monthly, we prepare supplemental indexes to these bulletins for library reference use only. These indexes are by alphabetical authors, original source code numbers and a sequential listing of the accession numbers. All these indexes are complete with titles. Any or all of the machine-produced indexes may be cumulated quarterly or as required.

One of the unique factors in the ITIRC system is the computer logic used for retrospective searching and current dissemination. It is a flexible technique for searching normal text data bases (it should be noted that the majority of our files are composed of author abstracts). The three experienced information retrieval specialists have achieved a high degree of precision with this search logic. As explained, the data base contains the language of the original document (normal text), and we can phrase search questions or user profiles in the same kind of normal English or accepted technical language. We use single terms, phrases, and adjacent or associated words. We scan not only the abstract, but the title, author, source information and subject terms associated with the document. Serving the needs of the information retrieval user and a complete description of the search logic may be found in the paper written and presented at the 1966 Spring Joint Computer Conference in Boston, Massachusetts, April 26–28, 1966.

With this kind of logic, the human intervener is a critical link in the system. The search question or the user profile must be tailored to the individual user needs, and written to take advantage of changes in language and development of new technologies whose terms are common today, but were not used previously. The sophistication of this system design provides the information retrieval specialist with the necessary logical tools, so that he can devote his time to understanding the user's questions and preparing them for the computer.

In regard to evaluating our system performance, our sole objective here is in terms of the users' satisfaction with the results they receive, both as Current Information Selection customers and as search request customers. With over 1,700 profiled domestic users in our Current Information Selection program, it was essential that we establish a mechanized feedback system. Therefore, accompanying

each printed abstract sent to a user, is a matching Port-A-Punch response card. When he receives an abstract, he simply punches out the appropriate box and returns the card to us. He has a choice of the following reactions to a document:

1 Abstract of interest, document not needed...
2 Send copy of document...
3 Abstract of interest, have seen document before...
4 Abstract not relevant to my profile...
5 Comments—
(Here the user may suggest changes to his profile, change of address, etc).

These Port-A-Punch cards are run against a computer statistical program. This statistical program supplies a complete analysis of all returns for each individual user and for all users, with separate reports for each of the major data bases—(1) total notifications sent out, (2) number of response cards returned, (3) number of interest (with a breakdown into each of the three responses listed above), and (4) number and percentage not relevant. In addition, the program gives us several special listing: (5) users who receive no notifications in the current period, (6) any user who fails to return the response cards within a specified period, and (7) a list of users whose "not relevant" response exceeds a predetermined percentage.

With the aid of the foregoing statistics, the IR specialist or profile specialist can readily identify those users who do not appear to be receiving satisfactory results from the system. Profiles can be reviewed and personal contacts made for necessary revisions and updates of profiles. It is the responsibility of these IR specialists working with profiles to monitor any changes they make, to be sure that the revisions are to the satisfaction of the user, and that he is receiving desired document notifications.

CONCLUSION

In conclusion, and with an eye to the future, we have learned that normal text searching has demonstrated itself to be economically feasible to continuing usage over the past two years. We know that normal text searching is capable of accomplishing meaningful information retrieval searches, both in dissemination and retrospective search request programs. We are witnessing an ever-increasing data base, which means we must develop faster search times. With the concern of the user uppermost in our minds, we attempt to provide twenty-four hour service, with an emergency service when needed, for all search requests.

Both within and outside the IBM Corporation, the field of information retrieval will continue to require constant study, research and development. We feel that for these activities, one of the best environments may well be within the framework of a live operating system that continually recognizes the needs of the individual user.

The Ohio Bell business information system

by E. K. McCOY

The Ohio Bell Telephone Company
Cleveland, Ohio

INTRODUCTION

This paper is a general description of the Ohio Bell Telephone Company's Business Information System with particular emphases on the handling of customer requests for service. The more common term used in our business for this is a customer's service order. The basic philosophy of the Business Information System is to have a central file of all service and equipment that is accessible by all persons having a need for this information. The system is designed this way so that it will be able to provide more information in a shorter period of time for our internal operations and as a result provide a much better grade of service to our customers

The Business Information System as viewed by Ohio Bell is designed not just for management but as as implement or tool for the clerk, analyst, service representative, information operator, plant installer and others. The Business Information System described here is an overall project of the Ohio Bell Telephone Company and is not designed to serve as an exact model for duplication by others. Every Business Information System must be created and tailored to meet the individual needs of those concerned.

Shown on Chart No. 1 is a map of the state of Ohio and a map of the geographical area we call the Northern Division of Ohio. This is essentially the city of Cleveland and contiguous communities. This area comprises approximately 600,000 telephone accounts and is forty-five miles long and ten miles wide. At the present time, it is anticipated that one system will be required for this area.

Chart No. 2 gives the particular statistics for the city of Cleveland. As indicated in the previous paragraph, there are approximately 600,000 accounts with a total of 1,100,000 telephones. This is served by five administrative districts and approximately 8,000 telephone employees. There is an average of 2,100 customer requests per day and this ranges from a maximum of 3,500 on Monday to 1,500 on Friday. There are about 5,000,000 local calls per day and 116,000 long distance calls per day. The month has been divided into twenty billing periods and thus, we produce 30,000 customer bills per business day.

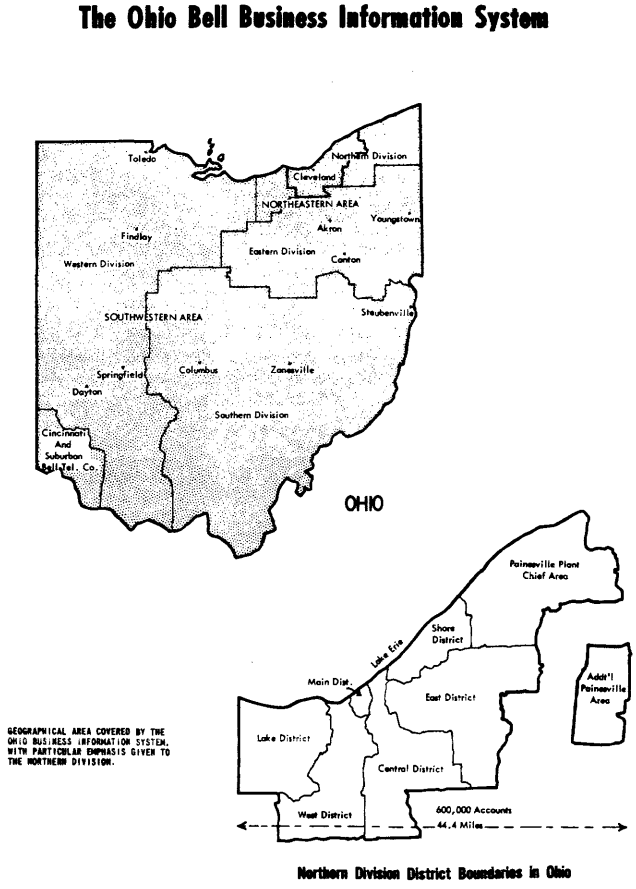


Chart 1 – The Ohio Bell business information system
Map of Ohio

Method of operation

Chart No. 3 shows the over-all general diagram of the Business Information System. To more readily

STATISTICS - NORTHERN DIVISION

ACCOUNTS - 590,000
TOTAL TELEPHONES - 1,100,000
ADMINISTRATIVE DISTRICTS - 5
TOTAL EMPLOYEES - 7,680
AVERAGE NUMBER CUSTOMER REQUESTS PER BUS. DAY - 2,100
LOCAL CALLS - AVERAGE PER BUS. DAY - 5,127,000
LONG DISTANCE CALLS - AVERAGE PER BUS. DAY - 116,000
BILLS ISSUED - 30,000 AVERAGE DAY

Chart 2 – Statistics – northern division

understand this system, I will trail a request by the customer through the many operations of our business. A customer may present his request for service to our business office service representative in person or by telephone. In either case, the request is then translated by the service representative to a coded format which is then sent via a model 35 teletypewriter directly into our computer.

When this teletype message is entered into the central computer, the computer will edit this message for any input errors that it may detect and proceed based upon what it detects. If an error is detected on input, a message is sent back to the sending station indicating the type of error detected. If the incoming message has passed the input edit successfully, it will then be processed by assigning a telephone number, switching equipment, and cable pair for Outside Plant Distribution to this order. The necessary information needed by the Plant Department to make connection between our central switching system and the outside plant has been sent to them via a teletypewriter. The Traffic Department also receives information at this time to update their information records. (This is shown by the solid black line in the upper right hand portion of this chart.) At the same time the Plant and Traffic Departments are receiving this information, the computer is sending a message to the Dispatch Center (indicated by black line to the left of center) giving them the necessary information for the installer to perform the necessary work at the customer's premises. Upon completion of this work, the installer will notify the Dispatch Center of his completion along with any additional information concerning the status of the installation. The Dispatch Center then notifies the computer again via a 35 teletypewriter (as shown by dotted line) that this service order has been completed either as submitted or as changed by the installer.

Since a copy of the initial order has been held by the computer in a Pending Order File, the Dispatch Center need only send to the computer that information required for completion of this order. When this

OHIO BELL
Business Information System

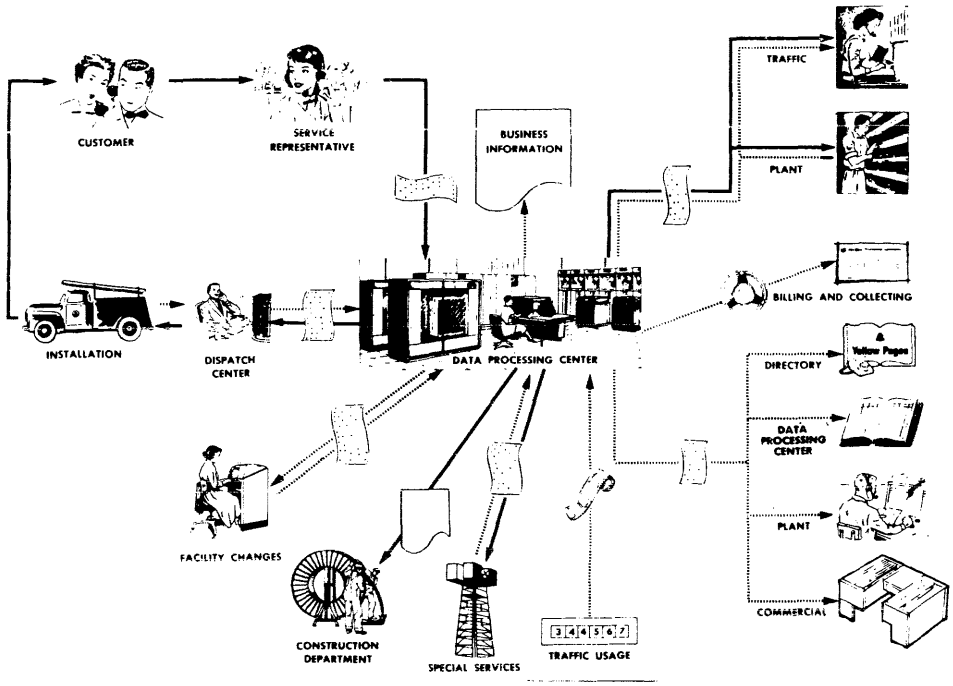


Chart 3 – Ohio Bell business information system
General description

notification is received by the computer, the computer will do all of the internal house keeping of the many files and provide the output required by the different departments served. The data associated with this order is removed from the Pending Order File at this time

Since the complete flow of all service requests has been established, the system is now in a position to make the many summaries of items that come from the service request. (This is indicated by the symbol in the center of the Business Information System Chart.) The dotted line in the lower right hand area of the chart indicates output from the computer to the Directory Department for the information required for the Yellow and White Page updating, data required by the Data Processing Center for accounting purposes, data to the Plant Repair Center so that an up-to-date file is kept on each customer's equipment and facilities, and a hard copy is sent to the Commercial service representative so that she has a complete copy of the customer's service. There is also a magnetic tape output from this system which will have the complete summary of a day's activity in a format that can be used as an entry to our Customer Records and Billing System which is on an IBM 7074 machine.

All of the files in the machine will be updated on a real-time basis via teletypewriter facilities.

At the bottom of this chart is shown Traffic Usage Data. Presently, this data comes from our many central office switching machines and provides the present means of measuring the load on our switching system. These data will be entered into our processing system and will be used as a basis for assigning new services in the proper central office unit.

Chart No. 4 shows in much more detail the schematic of the UNIVAC 491 system which is the heart of our Business Information System. The central processor (as shown in the center of the diagram) is a UNIVAC 491 with 32K core, 4 micro-second access time, memory guard and eight input/output channels.

Starting at the top of the diagram, we have used one channel for the internal clock which provides the information for printing the military time of day on all input/output messages.

The second channel is used for the console.

The third channel is used for handling six model 6C tape drives. These tape drives are compatible with the IBM 729 tape drives which are on the IBM 7074 system. These tape drives become the interface between the two systems.

The fourth channel is used for an output printer which can be either on-line or off-line. Presently, a UNIVAC 1004 card reader-printer is used on this channel.

The fifth and sixth channels are used for the communications system. Presently, we are using only one of these, the other is reserved for communications net-

OHIO BELL
Business Information System
Schematic of UNIVAC 491 Configuration

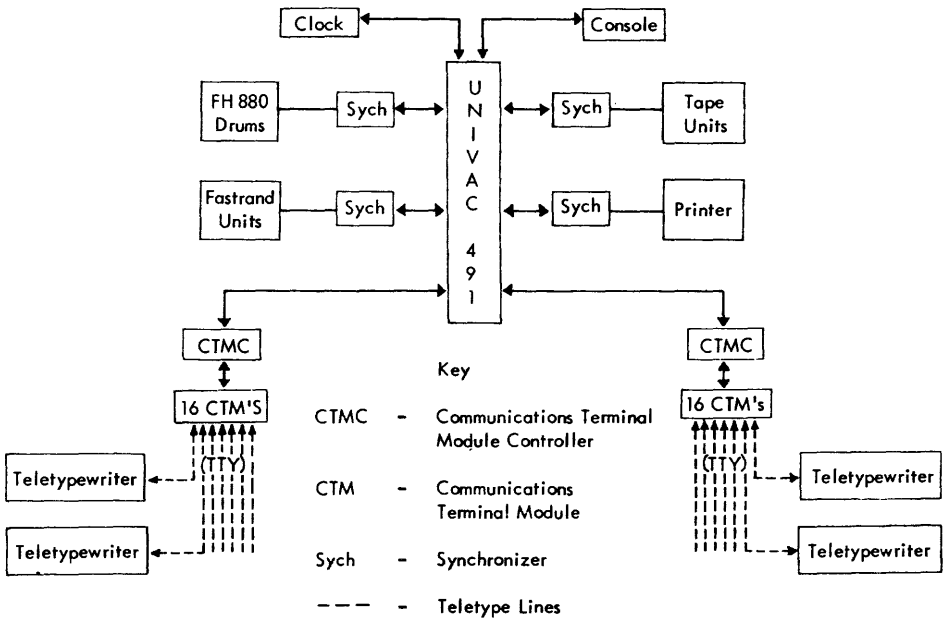


Chart 4—Schematic of UNIVAC 491 configuration

work growth.

Our communications system, as indicated previously, is made up of model 35 type teletypewriters and has been designed so that we can go up to six teletypewriters per line with individual call directing codes. Each line is terminated on a communications module. These sixteen communication modules can serve up to sixty-four two-way lines. The communications terminal module controller is a polling system that polls each of the teletypewriters every six seconds.
The communications terminal module controller is a polling system that polls each of the teletypewriters every six seconds.

The seventh channel is used for the mass storage device which in this system is a Model II Fastrand. This is a drum unit with a capacity of 120 million characters and an average access time of 90 milli-seconds. We can add up to eight fastrands per channel but our estimate at this time indicates it will take four of these units to handle the 600,000 accounts.

The eighth channel is used for two FH880 drums. Each of these drums have a capacity of four million characters and an average access time of 17 milliseconds. The fastrands are used to contain the master file of the customer's equipment, central office equipment and the outside plant facilities. We use the FH880 drums to contain our worker programs for the many different types of order processing. We also use these drums for queuing of input and output messages from the communications system. Messages are drawn from this queue on a predetermined priority level for processing. After processing has been completed, the messages are again queued for output to the communications system and again a priority level is used in this distribution.

Chart No. 5 shows a teletypewriter line to a remote

OHIO BELL
BUSINESS INFORMATION SYSTEM

TYPICAL TELETYPE CIRCUIT

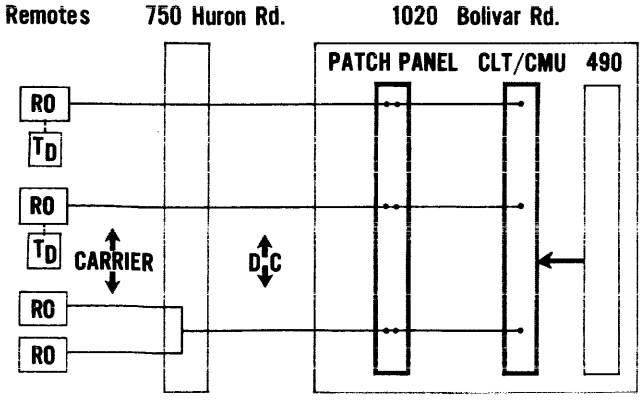


Chart 5 – Typical teletype circuit

machine which may be either a transmitter distributor machine which may be either a transmitter distributor, receive only, or automatic send and receive (ASR). These machines are served via a 43A carrier system from the downtown central office switching center at 750 Huron Road. From this point to our computer center which is at 1020 Bolivar Road, a distance of approximately 2,000 feet, we use DC lines for each teletypewriter circuit. These DC lines are terminated in a patch panel at the computer site. This patch panel is installed for ease of testing since we may test inward or outward from the computer or monitor a particular line for a service reason. This patch panel serves as the interface between the outside communications and the 491 computer terminals. The communications terminal module and communications terminal module controller is the first piece of computer gear contacted by the outside lines. From this point to the UNIVAC 491 there is a high speed channel (indicated on Chart No. 4).

Chart No. 6 lists the functions of BELLCON for the communications system.

OHIO BELL
BUSINESS INFORMATION SYSTEM

FUNCTIONS OF BELLCON-TELETYPE NETWORK

1. POLLING
2. EDITING
3. VALIDITY CHECKING
4. QUEUING
5. RECORD OF INVALID MESSAGES
6. RETAINED OUTPUT MESSAGES
7. AUTOMATIC OUT-OF-ORDER CONDITION

Chart 6 – Functions of Bellcon-teletype network

1. Controls the polling of different stations. This polling interval can be changed by a program change.
2. Performs an input edit for invalid characters.
3. Priority checks on the input to the computer are made by this program. As indicated in the schematic chart, the queuing of input messages is handled by this program.
4. A count of invalid messages is kept so that at the termination of a particular day the input station receives a count of messages transmitted.
5. A record of all messages is kept for one hour so that any operator can obtain a repeat message if it is required.
6. An automatic out-of-order message is sent to the communications center at the computer location

if any input station has a problem. This message is received by the communications supervisor and a decision is made concerning repair action.

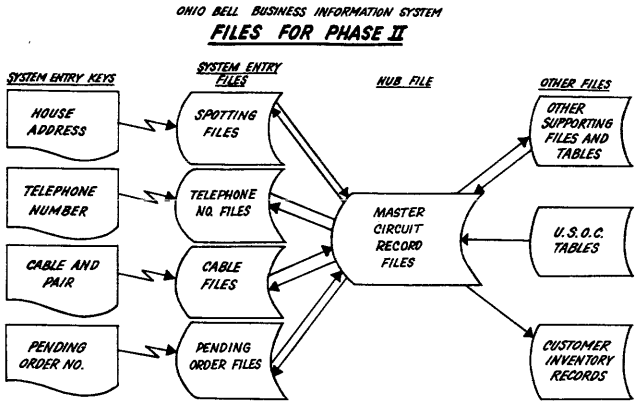


Chart 7 – Files for Phase II

Chart No. 7 shows a general schematic of our basic file design. This has been designed utilizing the philosophy of a Hub File. The Hub File in this system is called a Master Circuit File. On the left is shown a system entry key and adjacent the System Entry File. The Spotting File is a file of house addresses arranged in numerical order and the Cable Pair File is arranged by cable number and numerically by pairs in a particular cable. The Pending Order File is arranged in numerical order but the Customer Inventory Record (CIR), which we call the Customer File, is arranged in random order. This file is a summary of the customer's equipment, type of service, name, monthly rate, and many other items of information that make up our customer's record.

The Universal Service Order Code (USOC) is a file that lists each type of equipment that a particular customer has.

One of the most important parts of this Real-Time Random Access System is the control system required to maintain the many files and handle the input requests on a concurrent basis. The basic 491 system has a Real-Time Executive (REX) control program provided by the hardware vendor. This system is basically a hardware handler and requires approximately 5,000 words in core. This system has been supplemented by the BELLCON (Bell-Control) which has developed into a control system of 5,000 words. Thus, the total control program required is REX plus BELLCON which is 10,000 words.

Charts Nos. 8 and 9 list some of the items in our control system:

1. Screens and regulates all worker program requests and submits these requests to the hardware handler in REX. This includes the magnetic tape system, FH880 drums where the worker programs are stored.

BELLCON

1. SCREENS AND REGULATES ALL WORKER PROGRAM HARDWARE REQUESTS AND SUBMITS THESE REQUESTS TO THE HARDWARE HANDLERS IN THE REAL-TIME EXECUTIVE SYSTEM (REX).

MAGNETIC TAPE SUBSYSTEM
FASTRAND DRUMS
FH-880 DRUMS

Chart 8 – Bellcon

BELLCON

(Continued)

2. CONTROLS THE COMMUNICATIONS NETWORK.

3. SETS UP AND CONTROLS THE ENVIRONMENT WITHIN WHICH THE WORKER PROGRAMS PROCESS.

4. PROTECTS THE SYSTEM THROUGH THE CHECKS THAT IT MAKES DURING THE COURSE OF THE PROCESSING.

Chart 9 – Bellcon

2. As indicated in an earlier chart, this control system also handles the communications network.
3. It sets up and controls the environment within which the worker programs are processed. This includes the accessibility of the many files and restoring the files to their regular condition.
4. It also protects the overall system through the many checks that it makes during the course of processing. Again, one of the basic reasons for file protection is that they are complete and accurate.

Implementation

The system that has just been outlined is a very large undertaking from the standpoint of the overall development and programming, and also from the standpoint of implementation. A system of this magnitude is too large to implement all at one time so it was divided into logical implementation phases. These phases are shown in Chart Nos. 10, 11 and 12.

Chart No. 10 shows the outline for Phases I and II.

Phase I is basically made up of two large batch jobs that were written for this machine so that the savings that could be obtained from them would be realized

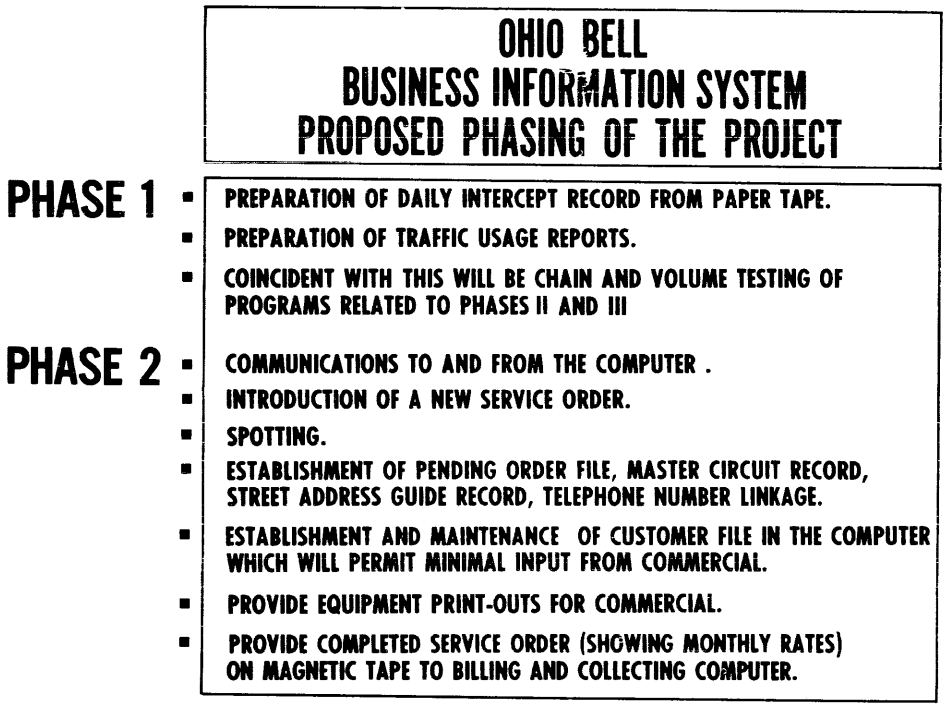


Chart 10 – Proposed phasing of the project

while the machine was being utilized to test later phases as they were programmed and available for testing and debugging.

The first job was that of preparing the daily intercept record used by the Traffic operator in handling calls concerning changes in telephone numbers. The second project was the use of the large computer to summarize and analyze the usage report obtained from the central office switching system.

Phase II is the first major step into the general Business Information System. This establishes a new service order format which includes the new Bell System Universal Service Order and at the same time established the 35 type teletypewriter communications network. The term "spotting" as used here means the selection of the outside plant distribution terminal that serves the particular address of the proposed customer. The rest of the items as shown on this chart pertain to the establishment of a basic record of the customer's equipment and his monthly rate for service.

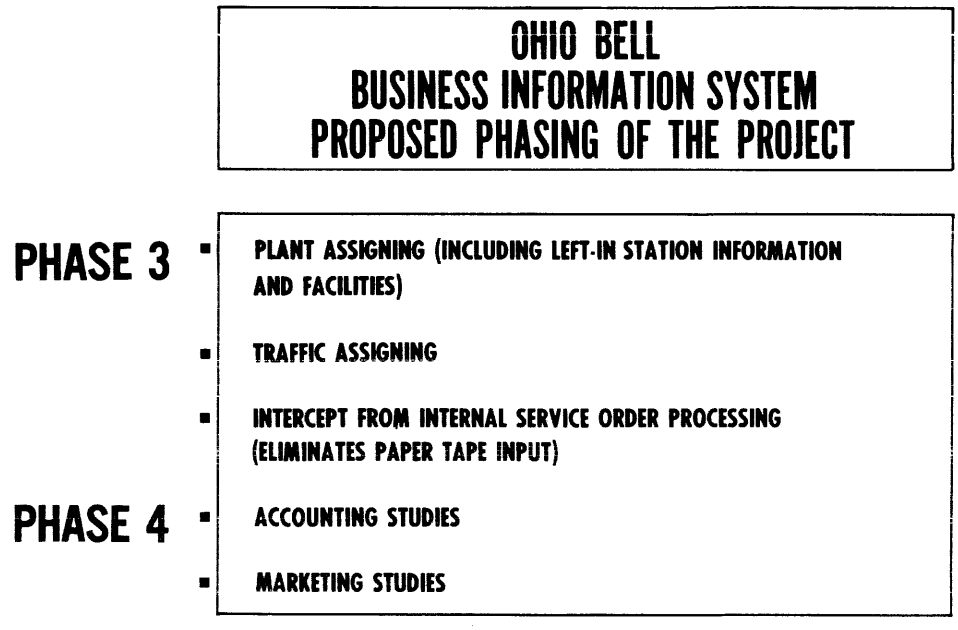


Chart 11 – Proposed phasing of the project

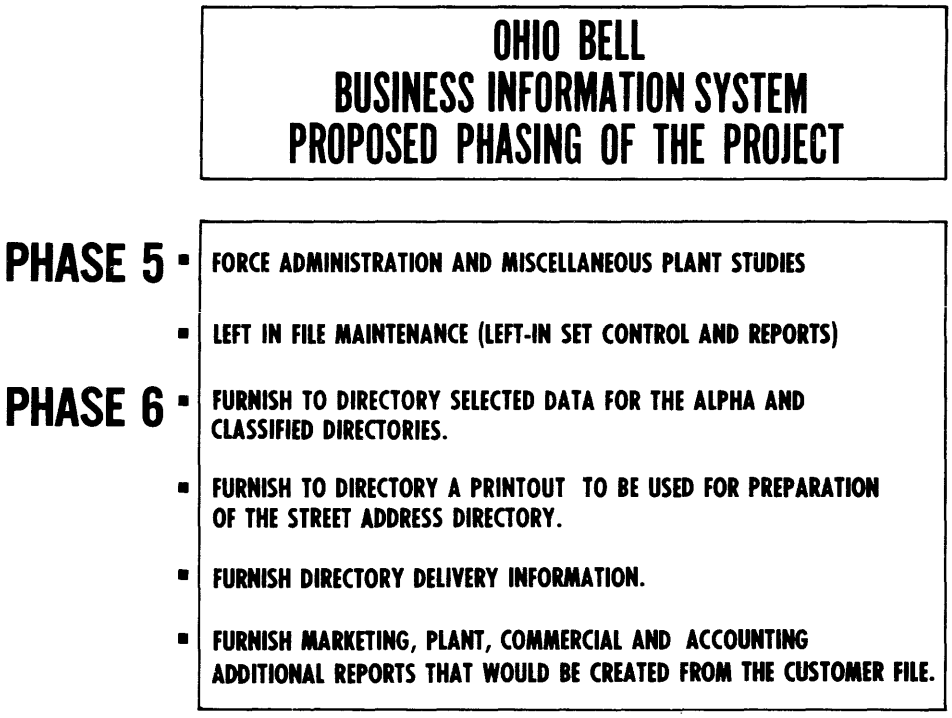


Chart 12—Proposed phasing of the project

Chart No. 11 shows Phases III and IV.

Phase III includes the computer assignment of all outside plant facilities and the computer assigning of all central office switching equipment, mainly, line and terminal equipment.

In Phase IV, the system is now in a position to make some of the many accounting and marketing studies that are required for the administration of the business from the flow of the customer's service request.

The last Chart No. 12 shows Phases V and VI.

In Phase V force administration data will be provided to the Plant Department giving that department the amount of work load building up for the days ahead. This work load can be arranged in any manner that the Plant Department requires. This phase also includes the file of left-in stations on subscribers premises.

In general Phase VI provides the many summaries required by the Directory Department for the Alphabetical and Classified Directories. It is also anticipated that this phase may become the input to PHOTAC (Photographic, Typesetting and Composing).

The complete implementation of these six phases for the Northern Division composed of 600,000 accounts will take approximately two and one-half years. Upon completion of this system, the many operating departments of our company will have access to all of the records contained in the machine and will have a tool that will enable them to provide the telephone service requested by the customer where he wants it, when he wants it, and the way he wants it.

Programming by questionnaire

by ALLEN S. GINSBERG
The RAND Corporation
Santa Monica, California

and

HARRY M. MARKOWITZ

California Analysis Center, Incorporated
Santa Monica, California

and

PAULA M. OLDFATHER
The RAND Corporation
Santa Monica, California

INTRODUCTION

The programming burden has often impeded computer application, but programming time and cost have been considerably reduced by the development of advanced programming languages such as FORTRAN, COBOL, and SIMSCRIPT. The objective of the technique discussed here is to further reduce the time and effort required to produce large computer programs within specified areas.

Programming by Questionnaire, or the Program Generator technique, is a method by which numerous programs in a given area can be constructed as quickly and easily as can a few large programs. This is done by bringing together the four components that form a Program Generator:

1 A Questionnaire, written in English, defining the scope and logic of all of the programs to be generated.
2 A Statement List containing all the computer commands needed to construct any of the many programs.
3 A set of Decision Tables specifying the commands required from the Statement List as a function of the Questionnaire choices.
4 The Editor Program for processing the Questionnaire, Statement List, and Decision Tables, thus building the desired program and providing the user with a list of the data he must supply to use the program.

Of these, only the Editor is general-purpose; the other three are specific to a given area of application. An application area consists of a family of programs centered around a single basic model that has many variations. All relevant details of these variations are presented to the user in the form of a multiple-choice Questionnaire. By choosing from the options, written in English, a nonprogramming analyst can specify all the information necessary to construct his program. He then submits his choices to the Editor, along with the Statement List and the Decision Tables. The Editor constructs the program, and supplies instructions for its execution and specifications for the data required.

The Job Shop Simulation Program Generator (JSSPG) was developed to demonstrate Programming by Questionnaire. The Questionnaire for the JSSPG consists of a booklet explaining 147 options in some detail, and an answer sheet (see Figure 1). These

CONDITION STUB	CONDITIONS
(Question Numbers)	(Answers to Questions)
ACTION STUB	ACTIONS
(Identification Numbers)	(Selected Statements)

Figure 1 – The JSSPG questionnaire answer sheet

options concern arrivals, routing and process times of jobs, decision rules for job dispatch and resource selection, shift changes, absenteeism, etc. The user specifies which options apply to the model he wants to generate. His choices, when given to the Editor along with the Statement List and Decision Tables, are translated into a SIMSCRIPT computer program.

Many different simulations can be generated in this manner. While there are not a full 2^{147} different models, since not every possible combination of answers is permissible, at least 2^{30} models are possible —more than one billion. The work required to develop the JSSPG, however, was comparable to the effort required to build a few large models.

General concepts

The questionnaire

The Questionnaire is the only part of a Program Generator the user sees. He does not need to know anything about the other components, but only to understand what he wishes to model, answer the Questionnaire, and supply input data. One important characteristic of the Generator concept is that the user must specify only the basic structure of his model and a few numerical data on the Questionnaire; he supplies the rest of the data when the Generated Program is executed. The Editor specifies the type and form of these data since they will vary from program to program, depending upon the options chosen on the Questionnaire. The only numbers required on the Questionnaire are those needed for storage allocation.

The statement list

The Statement List consists of all the commands, partial commands, groups of commands, and special instructions required to build any program that may be described on the Questionnaire. Since the Editor produces SIMSCRIPT programs, the Statement List also includes all the information peculiar to SIMSCRIPT, such as definitions of variables and initialization data, and special information required in the Generated Program.

The commands in the Statement List could have been written in any computer language, from machine language to the higher languages such as COBOL and SIMSCRIPT. Since the effort involved in building a Program Generator depends largely on the length and complexity of the Statement List, a Generator is easier to build using a higher language because of its simplicity, flexibility, and ability to accomplish a task with fewer commands. If any language other than SIMSCRIPT were used, the Editor would require modifications whose extent would depend upon the complexity of the language and its similarity to SIMSCRIPT.

The commands look exactly as they would in any other program, except for their Identification Numbers. If a command has no Identification Number, it is considered part of the preceding command that does have a Number. Thus, whole groups of commands can be called from the Statement List by simply selecting one Identification Number.

A Statement List command may also differ from a normal command in that it may be only part of a complete command. For example, in constructing the Statement List for the JSSPG, it was found that a given phrase (such as FIND FIRST) could be controlled by a variety of control phrases (such as FOR EACH JOB), depending upon which options the user chose on the Questionnaire. Rather than repeat the phrase common to all options, it is convenient to give it an Identification Number and number each of its modifiers. The Editor then combines all parts into a single command.

In addition to commands, the Statement List contains other information. For the SIMSCRIPT language, this includes the definition cards that specify the program variables and the initialization cards that specify the array sizes in memory. Also, the user is told what data he must supply in order to use the Generated Program, and the computer is supplied with control cards for execution. All four of these components (the definition and initialization cards, the input data requirements and instructions for its format, and the control cards) are cards in the Statement List, just like the commands, and are selected, based on the user's answers to the Questionnaire, in the same manner as the commands.

Decision tables

The Editor employs the Decision Tables to decide, based on the responses to the Questionnaire, which cards in the Statement List to include in the Generated Program. In building the Statement List, the programmer must have in mind the various combinations of commands needed to satisfy all the options that the user may select on the Questionnaire. The Decision Tables are merely a formal statement of the relationship between the Questionnaire choices and the programming statements. They give the Editor a set of rules, in a standardized manner, so that it can choose the right statements and handle them properly.

Decision Tables are divided into four quadrants, usually named as in Figure 2. The entries in the Condition Stub are keyed to questions on the Questionnaire; those on the Action Stub correspond to Identification Numbers in the Statement List. Thus, the Decision Tables link the Questionnaire and the Statement List.

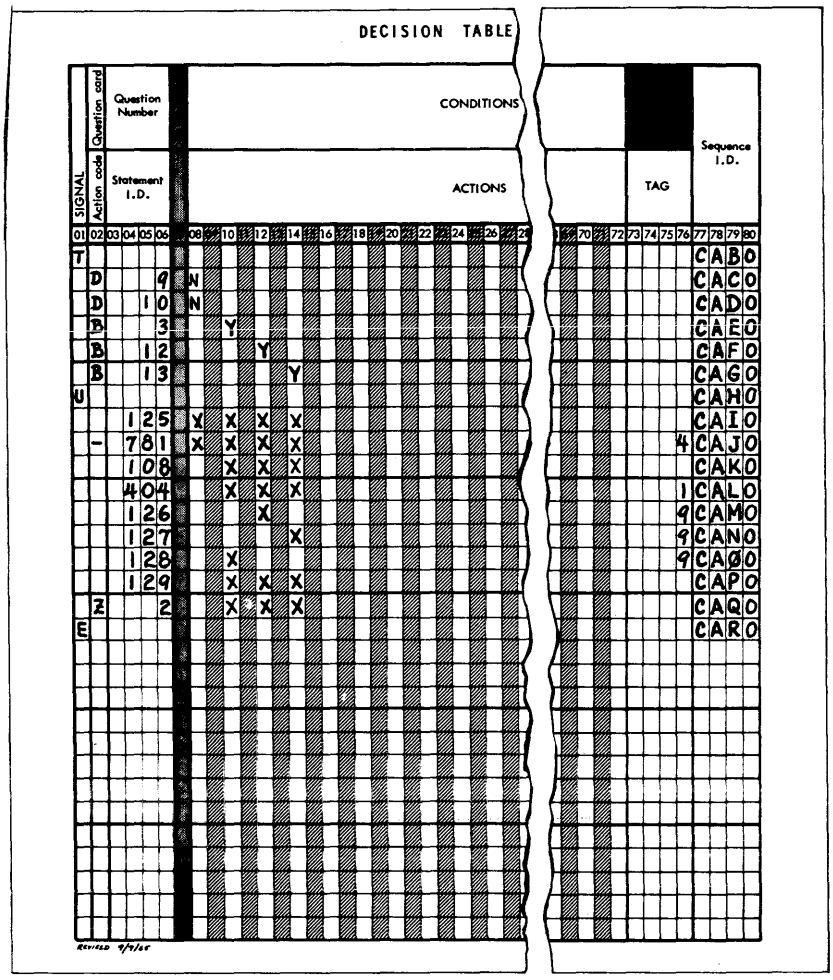


Figure 2 – Decision table terminology

The Conditions indicate the responses to the Questionnaire that are required for action to be taken. The Actions indicate which statements (whose Identification Numbers appear in the Action Stub) should be included in the Generated Program, given that certain questions or combinations of questions have been answered in a certain way.

Figure 3 shows a sample Decision Table from the JSSPG. Each column in the Conditions field represents a single combination of feasible responses to the questions listed on the Condition Stub. For example, there are four allowable combinations of responses to the five questions shown in Figure 3, as the nature of the questions precludes any other combinations. The first column of two N's would apply if neither D 9 nor D 10 on the Questionnaire was chosen, i.e., the user had, in effect, answered them both ((No." The second column indicates a "Yes" response to B 3 and apparent indifference to the other four questions. In actuality, the second column would apply only if either D 9 or D 10 had been answered "Yes." The column order is significant in that the first column found to match the answers on the Questionnaire, when scanning from left to right, will be the column that applies. Thus, the second column appears to imply indifference to D 9 and D 10, but it would only be chosen if both B 3 and either D 9 or D 10 were answered "Yes." Likewise the third column could be chosen only if either D 9 or D 10 was "Yes," B 3 was "No," and B 12 was "Yes." A similar statement could be made for the fourth column.

When a column matches the answers on the Questionnaire, the actions to be taken are found in the same column in the Action portion of the Table, indicated there by X's. For example, if neither D 9 nor D 10 was selected, the Editor would include the statements with Identification Numbers 125 and 781 in the Generated Program. The minus in front of statement 781 tells the Editor that card 781 is not a regular command, but rather is special information (in this case, a special definition card).

The fifth column in the Conditions can be reached only if the Questionnaire answers are inconsistent; i.e., no feasible combination of answers to the questions has been realized, implying that the user has

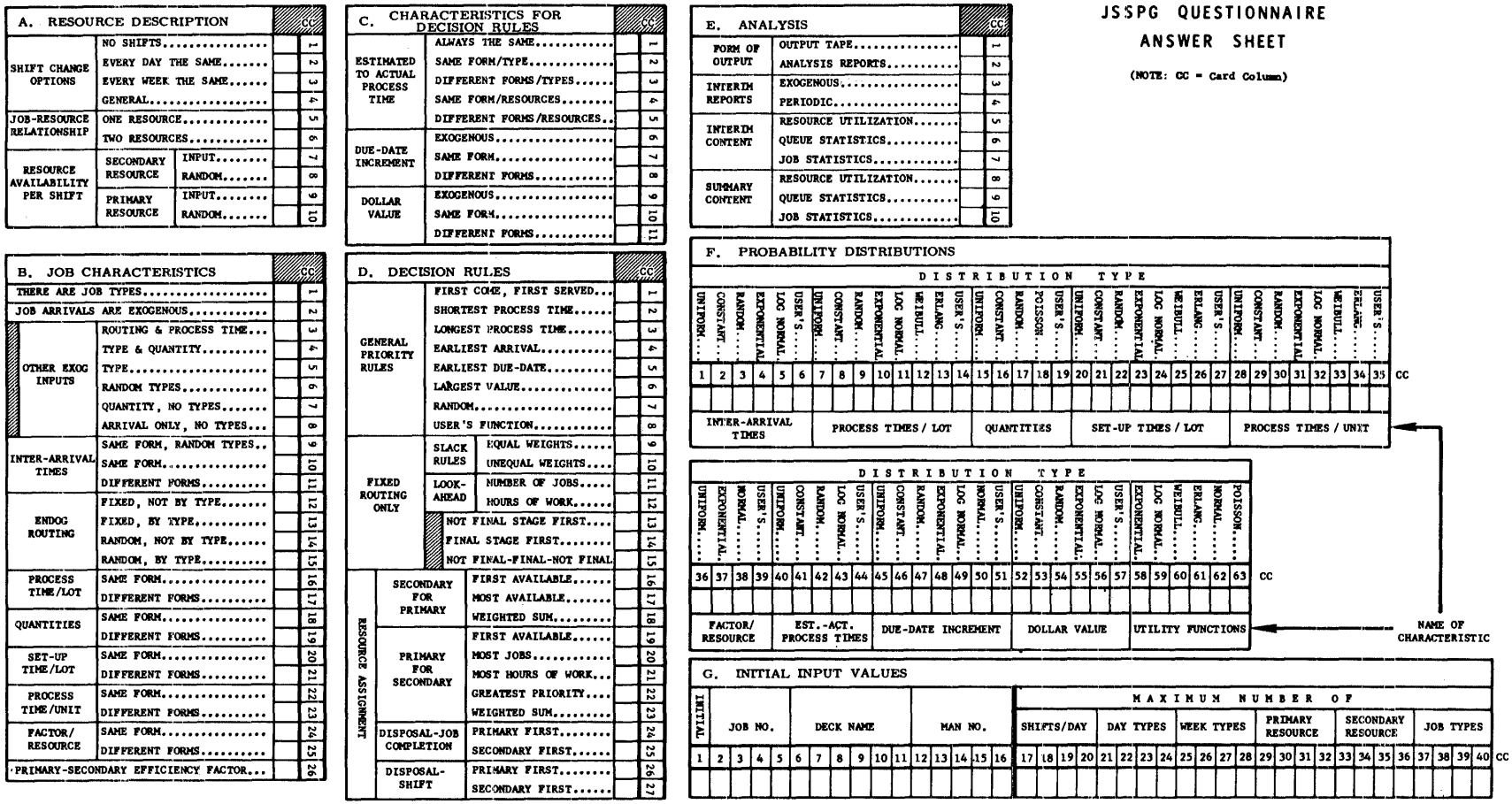


Figure 3 – Sample JSSPG decision table

made an error in completing the Questionnaire. The Editor, upon finding no Actions to be taken, rechecks the Conditions. If the Table indicates indifference to all questions, the Editor writes an error message, telling the user his responses are inconsistent. It also gives him a list of the questions involved and his responses to them. It is, however, permissible for there to be no Actions, or for there to be a final blank column reached in the Conditions by a process of elimination, thus implying apparent indifference to all questions, but not both.

The editor

In the broadest sense, the Editor is the computer program that translates the user's responses to the Questionnaire into a computer program. The Editor is written in SIMSCRIPT and is capable of producing only SIMSCRIPT programs of either a simulation or nonsimulation nature.

Operation

The Editor treats as input the other three parts of the Program Generator: the Questionnaire, the Statement List, and the Decision Tables. While their contents will differ with Program Generators written for different application areas, their logic and construction will be the same. Thus, the current version of the Editor can be used for other Program Generators in any area of application, if the language used for the Statement List is SIMSCRIPT, and if the principal components are constructed as outlined here.

The Editor has four functions:

1. To translate the answers to the Questionnaire into a computer program.
2. To check the answers to the Questionnaire for completeness and consistency.
3. To supply all control cards and dictionary information necessary to execute the Generated Program.
4. To provide a list of the names, meanings, and dimensions of the variables whose values the user must supply to the Generated Program during execution.

Besides a printed listing of the program, the Editor can supply a corresponding deck of cards that contains all the required control cards if the Program Generator is in the operational phase and not the development stage. The user need only place the required input data at the back of this deck where indicated and submit it to the computer for compilation and execution.

Other techniques

In addition to the obvious savings of time and effort, at least two other important benefits accrue from the automatic preparation of computer programs. Since available features are frequently presented in tabular form, they can serve as a checklist to remind the analyst of facets of his problem that he may have overlooked. Also, with a new or modified program readily obtainable at small cost, the analyst will frequently investigate less obvious alternative solutions that might not be considered if an existing program had to be modified. The desirability of these benefits has prompted many attempts at automatic program preparation.

A number of different techniques have been developed. In one, called the "Modular" approach, the user builds a program by selecting and linking a number of pre-programmed subroutines. This approach is often unworkable; it is difficult to make the subroutines logically compatible and the method necessitates a large library of subroutines. The approach may prove feasible if the set of options presented to a user is relatively small, such as in a sort program generator.* Because the Program Generator uses decision tables and compiles programs from individual commands rather than from larger modules, it alleviates most of these difficulties, allowing the user a much wider range of options.

The "Generalized Model" approach uses one large program containing all possible options; those to be executed in a given run are specified by parameter assignment in the input data.† The principal difficulty with this method is the inefficient use of computer time and memory space. The program must continually interrogate the parameters the user specified, using the results to decide which options are to be performed. If the options are very numerous, this process takes up a significant portion of the total running time. The Program Generator escapes this difficulty; it checks the options only once, at the time the program is generated.

The Generalized Model uses computer memory space inefficiently because the memory must contain all the options and the logic for choosing among them during the program's execution. Memory size therefore limits the number of options available. Since a Program Generator constructs only the code necessary to execute the chosen options, the generality

*See, for example, *IBM 7090 Generalized Sorting System*, IBM Systems Library, File No. 7090-33.

†One illustration is: *The Job Shop Simulator*, Mathematics and Applications Department, Data Systems Division, IBM.

of the approach is usually not limited by available memory space.

Program generators are not new. An example of a previous generator is the GQS.† As a consequence of how the user specifies his desired model and of the method used to generate the program, however, the range of options that can be offered in any one such compiler is very limited, as compared to the Questionnaire method.

†George W. Armerding, *General Single-Server Queue-Simulation Compiler*, Interim Technical Report No. 15, Fundamental Investigations in Methods of Operations Research, M.I.T., July 1960.

In summary, the following features of Programming by Questionnaire distinguish it from other methods of program generation:

1 An English language questionnaire, requiring no special knowledge of either detailed programming languages or format specifications.
2 Generation of computer programs that are as efficient as possible (given the limitations of both the language used for the Statement List and the person building the generator) in terms of computer memory space and computer time.
3 The ability to provide an almost unlimited range of options in any one generator.

Compiler generation using formal specification of procedure-oriented and machine languages

by PHILIP GILBERT
Measurement Analysis Corporation
Los Angeles, California

and

WILLIAM G. McLELLAN
Rome Air Development Center
Griffiss Air Force Base, New York

INTRODUCTION

This paper reports on a recently developed compiler generation system which is rigorously based, and which allows formal specification both of source (procedure-oriented) languages (POLs) and of machine languages (MLs). Concepts underlying the system are discussed, an example correlating source language specification with system operation is given, and the status and potentialities of the system are discussed.

The crucial problem of compiler generation is the characterization of procedure-oriented languages; the process is of limited use unless such characterization allows machine-independent processing of programs in these languages (and hence allows invariance of the language itself from machine to machine). Our solution interposes between POL and ML a "buffer" or "intermediate" language, called BASE, thus reducing the required POL→ML transformation to two logically independent subtransformations:

(1) POL→BASE (called *compilation*)
(2) BASE→ML (called *translation*)

This arrangement isolates questions of POL characterization within the first transformation, and questions of ML characterization within the second transformation. BASE itself is an expandable set of non-machine-specific operators,* declarators, etc., expressed in a uniform "functional" or "macro" notation; the meaning or intent of such operators is arbitrary insofar as the compilation transformation is concerned. The POL→BASE transformation may then be regarded as a machine-independent conversion, from a grammatically rich format to a simple linear format.

*Each BASE operation, declarator, etc., consists of a three-letter operation code followed by $n \geq 1$ operand type specifier/operand pairs S_i/X_i; e.g., $FFF(S_e/X_e, \ldots, S_n/X_n)$.

Theoretical basis

Within our system, a POL is characterized principally by a grammar (i.e., set of syntactic productions), and the consequent processing of programs in the POL is syntax-driven. To assure adequacy with respect to completeness, ambiguity, and finiteness of analysis, our snytactic method is rigorously based. A grammatical model (the *analytic grammar*) was developed, which provides a rigorous description of syntactic analysis via formalization of the notion of a scan. Within this model, the selection process of a scanning procedure can be precisely stated, and thus made amenable to theoretical investigation. Some characteristics of this model are:

- all analytic languages are recursive
- all recursive sets are analytic languages
- all phrase structure grammars are analytic grammars
- there is a simple sufficient condition under which an analytic grammar provides unique analyses for all strings.

The grammar in a POL specification permits certain abbreviations and orderings of productions (for convenience, brevity, and efficiency), but is never-

theless equivalent to a grammar using the simple scan S_4 of reference 8. (An equivalent grammar using S_4 is obtainable via a simple construction)) Context-sensitive productions may be used. Our method guarantees uniqueness of analysis—it is impossible to embed syntactic anbiguity in a language specification. A simple test ensures finite analyses of all strings. Such a grammar is at least as inclusive as the context-sensitive phrase structure grammar, and there does not appear to be any grammatical structure which cannot be accommodated (grammars of ALGOL, JOVIAL, and FORTRAN were obtained without difficulty).

In fact, such grammars are sufficiently powerful to accommodate the notions of "definition" and "counting" (cf. 7 and the examples of[8]), but to actually do so is neither efficient nor expedient. Therefore, a POL characterization includes description of pertinent "internal operations" (see the example in this paper).

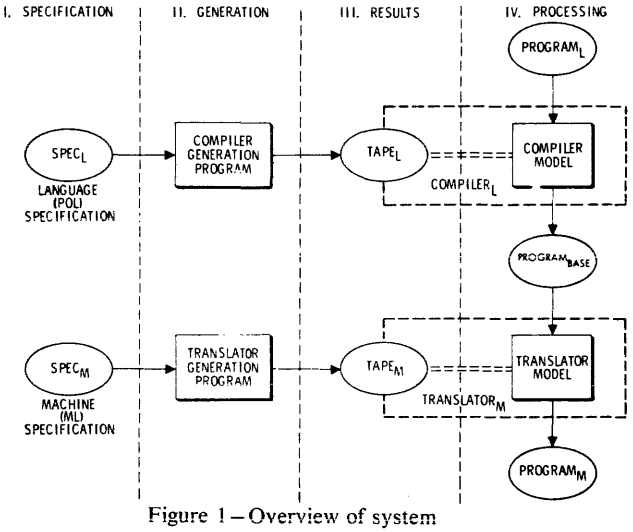


Figure 1—Overview of system

System overview

An overview of the generation system is shown in Figure 1. Using this system, the transformation from a source language L to a machine language M is achieved as follows:

A *specification* of L—an abstract description of the syntactic structure, "internal processing rules," and "output code" for L—is written. This specification is processed by the *compiler generation system* to produce a *tape* of L—a set of data tables corresponding to the specification. The compiler for L is then formed by conjunction of the tape of L with a *compiler model* program, a table-directed processor which acts simply as a machine for interpreting the tape of L.

Similarly, a specification of M is written designating macro-expansions appropriate to M. This specification is processed by a *translator generation system* to produce a tape of M—data tables containing the specified macro-expansions. The translator for M is then formed by conjunction of this tape with a *translator model* program, which expands BASE operations to sequences of instructions in M as directed by the tape of M.

Compilation system data base

Processing of input strings (POL programs) by a generated compiler is intended to occur in two parts:

(a) preliminary conversion of "raw" input symbols to yield a "syntactic" or "construct" string, which represents the raw input for all further processing, and then

(b) step-by-step syntactic analysis, and (at each analysis step) performance of prescribed sets of internal operations, prescribed output of "code blocks," output of diagnostic messages, and (if desired) performance of additional auxiliary processes.

The internal operations in a POL specification assume a set of data entities (the "date base"), which are later manipulated as prescribed by a generated compiler. Each entry of the *construct string* (which represents the raw input during processing) contains a *construct* (or *syntactic type* or token) and an associated datum, which is originally derived from the raw input, but may be internally altered. The use of appropriate string handling routines allows effectively a construct string of unbounded length. Other data entities are:

(a) a set of *function registers* F_i, for storage and manipulation of "temporary" numeric data

(b) a set of *symbol registers* S_i, for manipulation of symbol strings

(c) a *property table* of integer properties $P_i(J)$, for storage and manipulation of numeric data (e.g., number of dimensions) associated with "variables" in the input string. "Names" (i.e., contents of symbol registers) can be "defined" to the table to reserve table entries for associated data, and the table can be "searched." Defined names are placed in a *property table index*. The J^{th} table entry consists of four *properties* $P_0(J)$, $P_1(J)$, $P_2(J)$, $P_3(J)$. By convention, $P_0(J)$ is the syntactic class of the corresponding defined name.

See Figure 2 for further details.

POL specification and compilation system operation

The relation between a POL specification and the consequent compilation system processing is best shown via an example. Figure 3 shows a specification* of the language LEMA2 (first exhibited in Lema 2,[2] which consists of sentences having the form

$'A^mB^nA^mB^nCCC'$

where X^k signifies a sequence of k X's. Some sen-

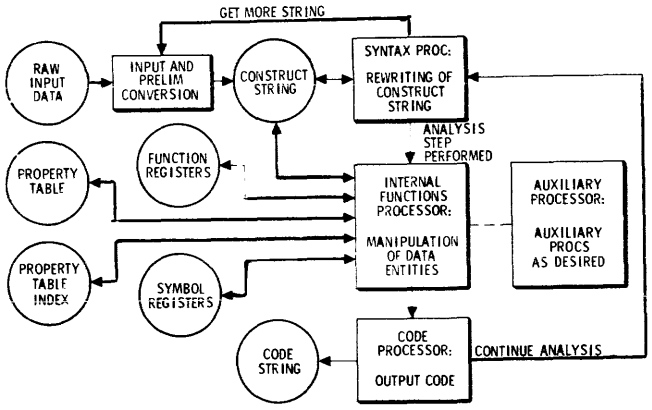


Figure 2 – Compiler model organization and system data entities

tences of LEMMA2 are

'AABBBAABBBCC'
'AAAABBAAAABBCCC'
'AAABBBBAAABBBBCCC'

The specification contains five sections:

(1) Symbols – specifies the preliminary conversion of input symbols and "reserved words" to construct string entires
(2) Syntax – a set of syntactic productions for use in syntactic analysis
(3) Internal Functions – the internal processing to be carried out at each analysis step
(4) Code – the sequences of codes to be output at each analysis step
(5) Diagnostic Messages – a set of messages for output

The sections containing internal functions, codes and diagnostic messages are unnecessary in defining the language structure, but have been added to illustrate these mechanisms. The codes BEG, PWR, AAA and BBB appearing in the code section were invented expressly for this example; arbitrary BASE operation codes may be designated at will, since these codes are merely transmitted during compilation. The following discussion can be correlated with Figure 4, which shows the compilation analysis trace for a LEMMA2 program, together with resulting values of function registers and code output at each analysis step.

The conversion specified in the Symbols section, of raw input symbols to construct string format, is performed specifically to eliminate dependency of processing on particular machine character sets and hollerith codes. A construct string entry containing a construct and an associated datum replaces each input symbol (or symbol sequence constituting a reserved word); Figure 5 illustrates this process. An arbitrary numeric or hollerith datum may be specified. Data

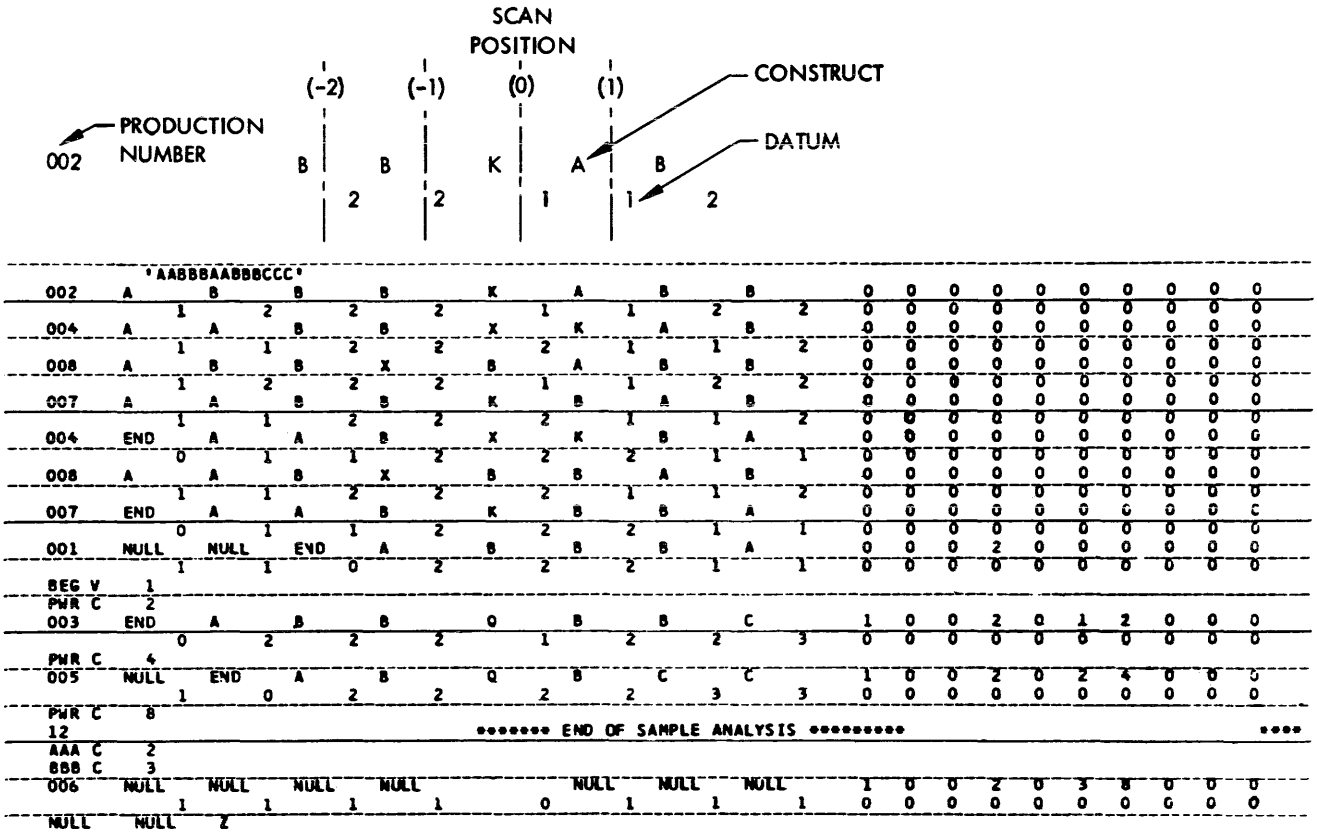
*The specification is shown in "reference" format, which differs trivially from the format used in machine processing of specifications.

```
      * TITLE (LEMMA2)
      * SYMBOLS
        (1)A          (A)       (1)
        (1)B          (B)       (2)
        (1)C          (C)       (3)
        (1)'          (END)     (0)
        (1)           (NULL)    (0)
        ((EOC))       (NULL)    (0)
      * END SYMBOLS
      * SYNTAX
001                   (A)(B)(K)   ==(B)
002                   (B)(A)(A)   ==(B)(K)(A)
003                   (B)(A)(B)   ==(Q)
004                   (B)(B)(K)   ==(B)(X)(K)
005                   (B)(Q)(B)   ==(Q)
006   (END)(A)(Q)(C)(C)(C)(END) ==(Z)
007                     (X)(B)    ==(K)(B)
008                     (X)(K)    ==(X)(B)
      * END SYNTAX
      * INTERNAL FUNCTIONS
001    RTV F3 -2
       INC F3 1
       ASQ -3 F3
003    SET F5 1
       SET F6 2
       PUT S1 V0(-1)
       SUF S1 V0(0)
       DEF S1 ((A))
       ASO 0 F0
       SET P1(F0) 1
005    INC F5 1
       MPY F6 2
006    PRN 1 S1
      * END INTERNAL FUNCTIONS
      * CODE
003    BEG(V/R(0))
       PWR(C/F6)
005    PWR(C/F6)
006    AAA(C/R(-5))
       BBB(C/F5)
      * END CODE
      * DIAGNOSTICS
0001 ******* END OF SAMPLE ANALYSIS **********
      * END DIAGNOSTICS
      * END DATA
```

Figure 3 – Specification of the LEMMA2 language

from the construct string may be used to construct symbol strings (names), but this usage is not dependent on the specific hollerith codes which are used.

The syntactic productions in a specification's Syntax section are applied (as determined by the compiler model's scan) to "rewrite" the construct string, in a step-by-step fashion (see Figure 6). The succession of these rewritings constitutes the syntactic analysis of the construct string. In selective productions from the set of Figure 6, the compiler model uses the "leftmost" scan S_1 of [8], i.e., at each step the production chosen is the one whose "left side" occurs first (leftmost) in the construct string. Thus at the first analysis step, the substring chosen is BAA;



- Each step of the analysis trace shows the string in the vicinity of the scan position, after application of the production, performance of internal functions, and code output.
- The code output for production p precedes the trace line for production p.
- Values of the first 20 function registers are shown at each analysis step: on line with construct F0 through F9, on line with datum F10 through F19.

Figure 4 – Compilation of a LEMMA2 program

at the second, ABK; and so on. To allow explicit reference to the data which accompany the constructs of the substring chosen, a *scan position* is defined (at each step) to occur at the last (rightmost) construct of the selected substring (see Figure 6).

At each analysis step, internal operations associated with the selected production are performed: function registers or properties within the property table may be set, used, or arithmetically manipulated; character strings may be placed in, prefixed to, or suffixed to symbol registers, and so on. The Internal Functions section (see Figure 7) consists of sequences of internal functions operations. The first operation of each sequence has the label of the production for which action is taken. Thus the sequence RTV F3 -2, etc., is performed each time production 001 is selected.

Care has been taken in formulating the internal operations to achieve economy of means – simple operations, a minimum of system data entities, and a minimum of compiler model machinery. Such a formulation allows a simple compiler model program, while language complexities must be expressed within the language specification. Some anomalies of notation still remain from our earlier efforts, but it is planned to revise and clarify notation.

Operation sequences pertaining to different productions are independent of each other, since there is no "GOTO" operation (a "skip forward" is sometimes permitted). Thus a finite sequence of operations is performed at any analysis step.

Code may be output at any analysis step. Operation codes and operand type specifiers given in the Code section (see Figure 8) are merely transferred to the output, while operands are inserted as specified.

The Diagnostic Message section contains a set of messages, which are output by PRN internal operations. The operation PRN1 S1, which is executed for production 006, prints message 001 and the contents of S1.

Translation and machine specification

A translator for a given target machine (ML) pro-

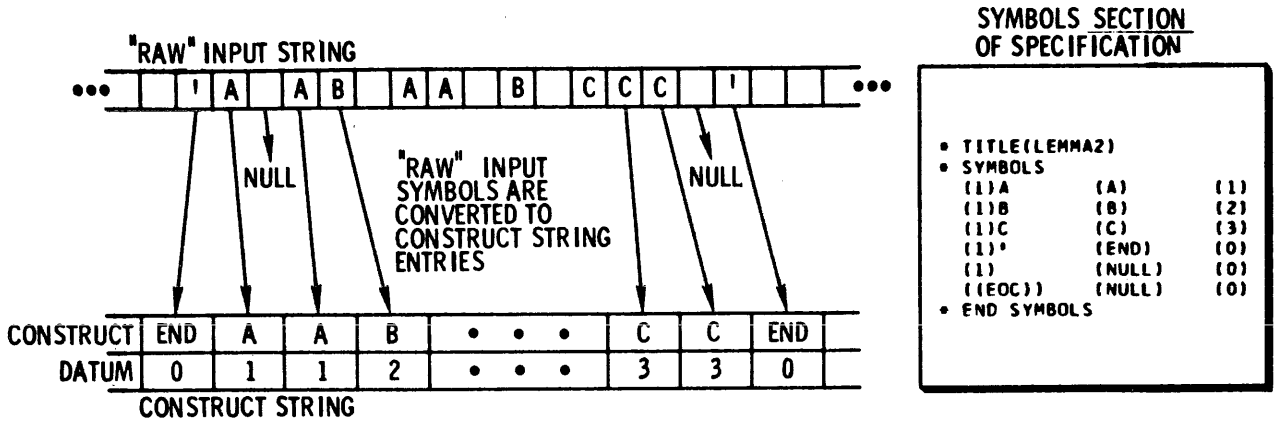


Figure 5. Preliminary Symbol Conversion

- The number in parentheses on the left indicates the number of characters comprising the reserved word. The symbols of the reserved word follow.
- A construct (e.g., (END)) is specified for each symbol or reserved word. Use of the construct (NULL) specifies that no construct string entry is to be made; thus "blanks" are ignored above.
- A datum is specified for each symbol or reserved word. Either a numeric datum (e.g., (3)) or a hollerith datum (e.g., ((h), where h is the desired hollerith datum) may be specified.
- The special notation ((EOC)) denotes the "end of card symbol", which in many languages is regarded as a punctuation mark. A representation of ((EOC)) must be given in every Symbols section.

Figure 5 – Preliminary symbol conversion

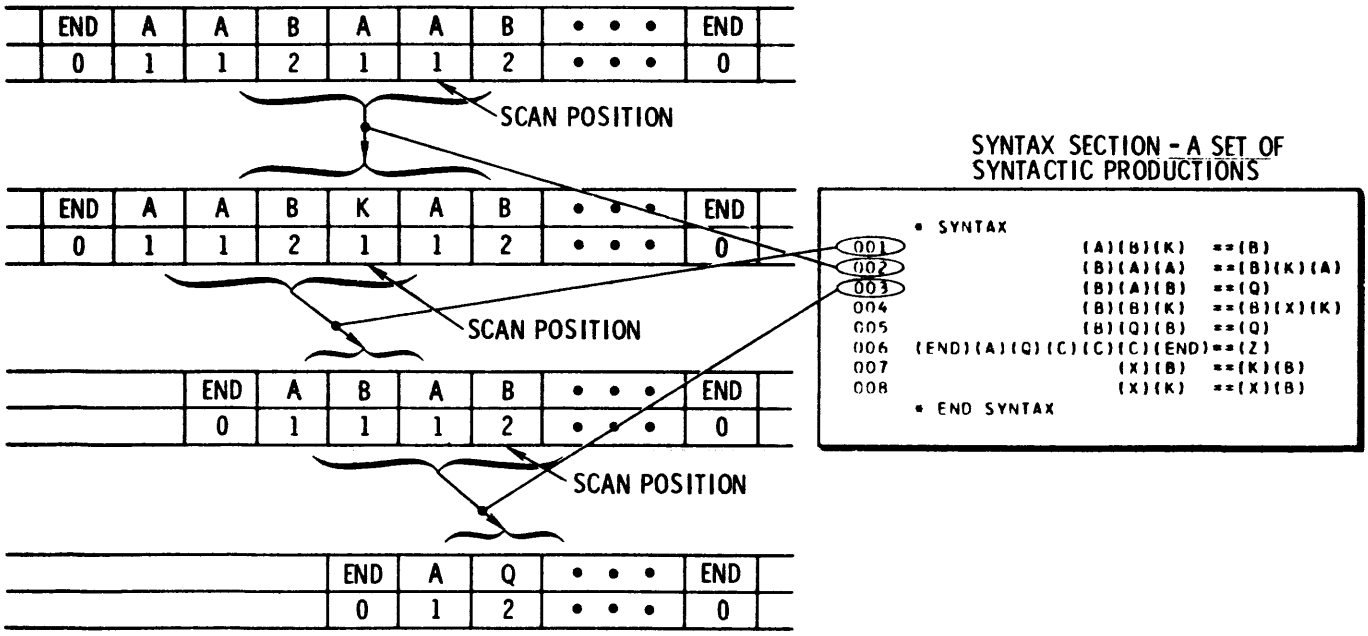


Figure 6 – Syntactic analysis

duces, from an input program of BASE operations, an equivalent program in the target assembly language, in a format acceptable to the target assembler. The production of assembly language guarantees compatibility of the object program with the machine's monitor system, and allows the assumption in translation of system subroutines and macros.

A BASE program contains generalized item declarators, array declarators, etc., and generalized computation operators (e.g., ADD, SUB). Since data definition is explicit, the BASE computation operators do not take account of the data types involved in the operations. Thus for each computation operation, there is an equivalent set of *standard suboperations*; e.g., corresponding to ADD are the standard suboperations

"add a floating item to a fixed item"
"add a fixed item to a floating item"

and so on. Determination of the specific suboperation required for a given BASE operation, taking into account the data types involved, is performed within the translator.

Translation thus occurs in two parts:

(a) analysis of BASE operations by an *analysis section*, to derive equivalent sequences of standard suboperations, followed by
(b) *expansion* of the standard suboperations by a *macro-processor section*, to produce assembly code.

A machine specification defines expansion of the standard suboperations. In other words, it defines for each standard suboperation an equivalent sequence of assembly language instructions. Embedded in these expansions are format specifiers, which cause the appropriate format to be generated. A machine specification is processed by the translator generation system to produce corresponding data tables, which are combined with the translator model program to form the desired translator. These data tables direct the expansions performed by the translator's macro-processor.

Parameters required by the expansions are furnished by the translator's analysis section via a communication table, from which they are retrieved as necessary by the macro-processor section. Within a machine specification, parameters are specified via position in this table.

Our present machine specification notation is processor-oriented, and not easily readable; however, it is planned to formalize this notation. Some typical macro definitions are shown in Figure 9, in a contemplated notation, as an illustration of the features provided in a machine specification.

The translator model program, except possibly for one output procedure, is machine-independent. The analysis of BASE operations is dependent only on the operator, accumulator data type, and operand data type involved, while macro expansion is table-driven. All dependency on the target machine is isolated within the data tables used to direct expansions. Assembly code is output in the form of 80 column card images, which are almost universally acceptable by target assemblers. Unusual cases might require simple modification of the output procedure.

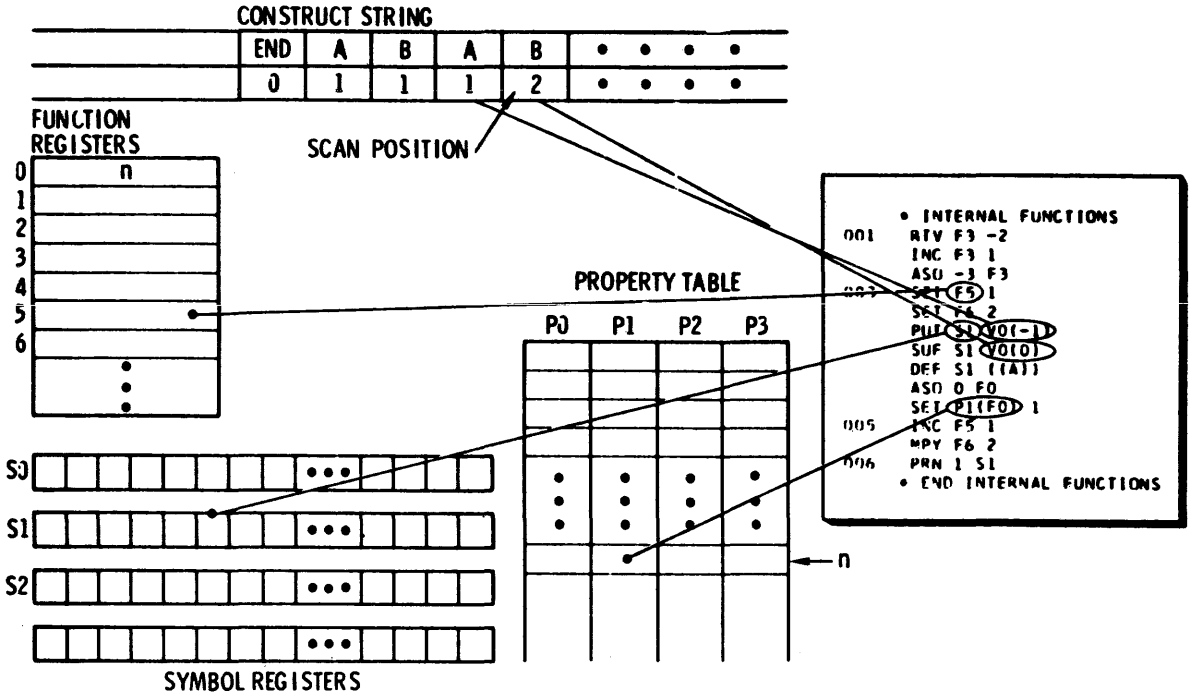
CONCLUSIONS

Using the syntactic model[8] we have developed a system to formally characterize languages which are rich in grammatical structure, and to subsequently process strings in such languages. Such processing can produce linear code (BASE language). The BASE language contains computation and data declaration operations sufficient to accommodate the functions of ALGOL, FORTRAN and JOVIAL. BASE is expandable, so that more convenient or efficient operations may be introduced when these are desirable. We have shown the feasibility of formally characterizing machine (assembly) language, and of machine-independent translation (BASE→ML). In sum, we have presented a rigorously based, machine-independent compiler generation system.

A consequence of these results is that language invariance can be maintained from machine to machine. It is possible to have a standard version of each procedure-oriented language, rather than machine-dependent variants.

The system is presently running on the CDC 1604 computer. Specifications of ALGOL, FORTRAN and JOVIAL have been written, as has machine specification for the CDC 1604. The ALGO and FORTRAN specifications have undergone tentative checkout and modification, as has the CDC 1604 specification. Preliminary comparisons of operating characteristics have been made. For a small number of short programs, our system produces object programs about the same size as do the manufacturer-supplied compilers, and requires between twice and three times the computer time. Since our system is a prototype, these results indicate that it may be possible to generate compiler/translator systems which have competitive efficiencies. We contemplate major operational changes, without the sacrifice of theoretical rigor, which should increase system speed by a factor of between 3 and 5.

The compiler (POL→BASE) portion of this system has other uses. The ability to formally characterize grammatically rich languages and to subsequently process strings in such languages is of importance wherever string-structure-dependent processing is required.



- SET F5 1 places the value 1 in the function register F5
- PUT S1 V0(-1) places the datum (regarded as hollerith) from construct string position (-1) - relative to the scan position - into the symbol register S1. All previous contents of S1 are deleted.
- SUF S1 V0(0) suffixes to the string in S1 the datum from construct string position 0.
- DEF S1 ((A)) "defines" the string in S1 to the property table: a property table entry (say the n^{th}) is reserved, the string in S1 is entered into the property table index, together with the entry number n. The number representing the construct (A) is placed in P0(n), and n is placed in F0.
- ASO 0 F0 "associates" the value in F0 with the construct in string position 0: the value F0 is placed in the datum of position 0.
- SET P1 (F0) 1 places the value 1 in P1(F0), i.e., in P1(n).

Figure 7 – Performance of internal function

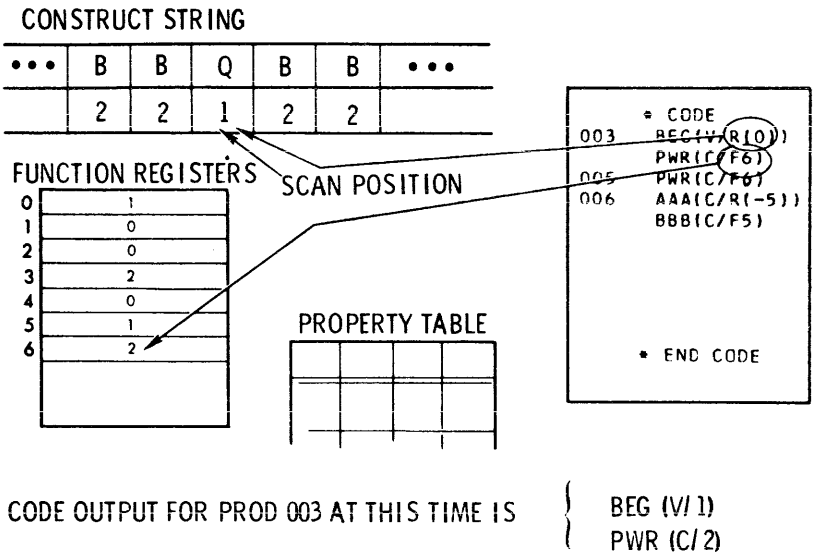
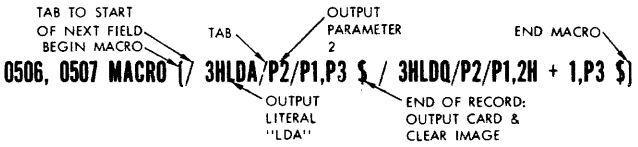


Figure 8 – Output of a code sequence

LOADING ACC AND MQ WITH DOUBLE PRECISION OR COMPLEX NUMBER (FOR CDC 1604):



LOAD ACCUMULATOR (FOR IBM 7094 FAP):

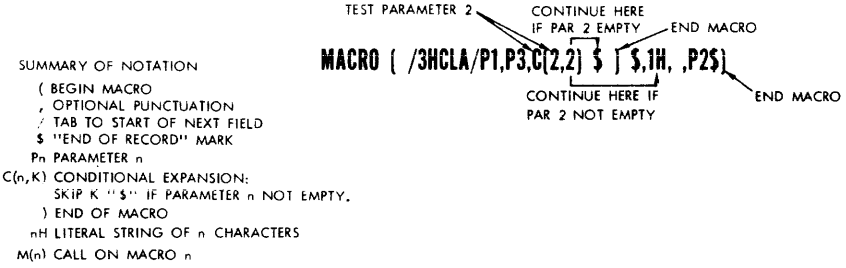


Figure 9 – Some typical macro definitions

ACKNOWLEDGMENT

The authors are indebted to Donald M. Gunn and Craig L. Schager, both for their significant contributions to this work and for their valuable suggestions regarding this paper.

REFERENCES

1. N CHOMSKY
Syntactic Structures
Mouton & Company The Hague 1957
2. N CHOMSKY
On certain formal properties of grammars
Inform Contr 2 137-167 1959
3. S GINSBURG and H G RICH
Two families of languages related to ALGOL
J ACM 9 350-371 July 1962
4. P NAUR (Ed)
Report on the algorithmic language ALGOL 60
Comm ACM 4 299-314 May 1960
5. E T IRONS
A syntax-directed compiler for ALGOL 60
Comm ACM 5 51-55 Jan 1961
6. D G CANTOR
On the ambiguity problem of Backus Systems
J ACM 9 477-479 Oct 1962
7. A C DiFORINO
Some remarks on the syntax of symbolic programming languages
Comm ACM 6 456-460 Aug 1963
8. P GILBERT
On the syntax of algorithmic languages
J ACM 13 90-107 Jan 1966
9. GILBERT, HOSLER and SCHAGER
Automatic programming techniques

RADC-TDR-62-632 Final Report for contract AF30(602)-2400

10 GILBERT, GUNN and SCHAGER
Automatic programming techniques
RADC-TR-66-54 Final Report for contract AF30(602)-3330

11 GILBERT, GUNN, SCHAGER and TESTERMAN
Automatic programming techniques
RADC-TR-66-665 Supplemental Report for contract AF30 (602)-3330

An experimental general purpose compiler

by RICHARD S. BANDAT and ROBERT L. WILKINS

Western Union Telegraph Company
Paramus, New Jersey

INTRODUCTION

With the advent of numerous programming languages, both special and general purpose, much interest has been generated in developing newer and higher level programming languages. Two notable approaches[1,2] have been taken in the attempt to provide language processors for the development of new programming languages with a minimum investment in programmer time and effort. One method is to provide, in an existing programming language, facilities for list processing, stack maintenance, and character manipulation, which can then be used to write a compiler for the language under development. Another approach is the now classic "Meta-Compiler" method typified by the "Meta" series of programs by Val Schorre et al. In these programs, a description of the syntax of the source language is given, together with a paraform code which contains transformational rules. The output is the desired compiler.

This paper describes a third approach which is sufficiently different so as not to be categorized as a subset of the above methods in design, but a union of their favorable characteristics. Its aim is to facilitate defining the syntax of new programming languages and to parse them so that there need be only one output routine for each operator in the new programming language.

Discussion

Our approach was as follows: First, implement a parsing program which would only require the hierarchal level of an item as input and yield an operator and its operands as output; second, design a generic method for determining the hierarchy and syntactic legality of each input character.

The parsing program can be viewed as a modified stacker and is covered later in the paper. An initial solution for the remaining problem was a matrix similar to the type used in a table-driven compiler.[3] However, it was noted that a very large matrix would be required for most useful languages because of their numerous syntactic types. In addition, it was also noted that there were two undesirable characteristics of this method; namely, large amounts of core were wasted because the matrix was sparse, and no efficient way could be devised to include transformational rules. The need for the latter capability directed our efforts toward another table which was neither sparse nor lacking transformational characteristics, yet which was sufficiently analyzed so as to enable its usage to be understood. The result was a table based on a Turing Machine.[4]

The first attempt towards a boot-strap compiler was made by filling in the Turing Table numerically by hand so that at least a rudimentary symbolic Meta-language program could be read in as source statements. For testing purposes, the symbolic Meta-language program described itself, or actually a Turing Table which would allow the syntax/solidus parser to read in its own meta-language. Naturally, the semantics routines were written so that their output would be the Turing Table for the language which was described by the meta-language. After "several" attempts a clean table was obtained which would read its description and generate an exact replica. This procedure was used to check out each version of the Meta-language, which then underwent continual revision in an attempt to increase its readability and raise its scope above the single input character level.

The procedure then for using the program as a compiler consists of two steps; namely, to describe the programming language and to write for each operator in that language a routine to accomplish the task for the type of compiler being written, i.e. interpretive or batch.

The overall size of this program, which is implemented in GE-235 Time Sharing FORTRAN, can be determined from the amount of code generated by its FORTRAN compiler. For the meta-compiler

version (i.e. semantics routines generating a Turning Table), 2.5K is used for instructions and about .5K minimum is required for all the tables. Coding the program in assembly language would surely reduce the figure by a significant amount.

Detailed description

The program consists of a small driver, Figure 1, three functional programs, and four utility routines, which include an internal symbol table. The driver initializes the routines and passes information between the main routines and the symbol table.

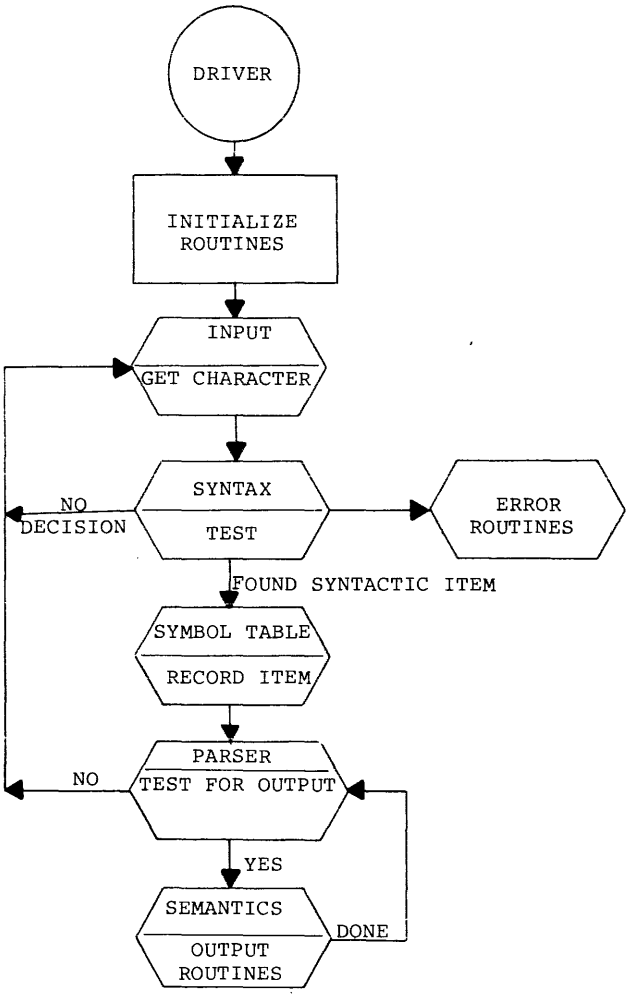


Figure 1 – Driver

Utility routines

The Input Program, Figure 2, simply reads one character at a time from the input buffer, refilling the buffer as needed.

The Classing Function, Figure 3, compares the input character with the table of legal input symbols and assigns to the character the class number which was assigned to it when the language was defined.

Since our present computer is the **GE-235 Time Sharing System,** it has been convenient to have a pack

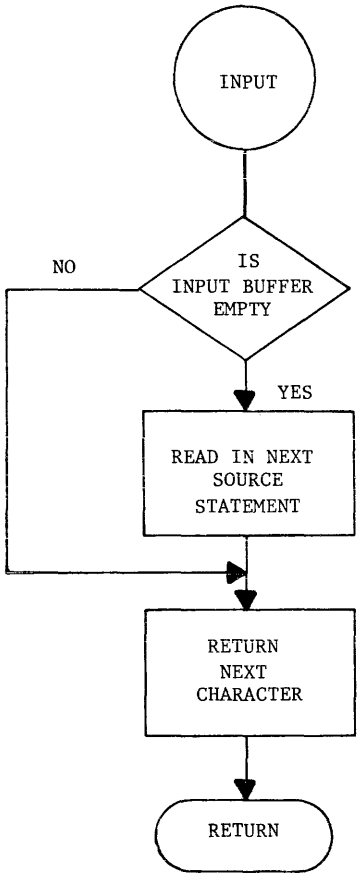


Figure 2 – Input function

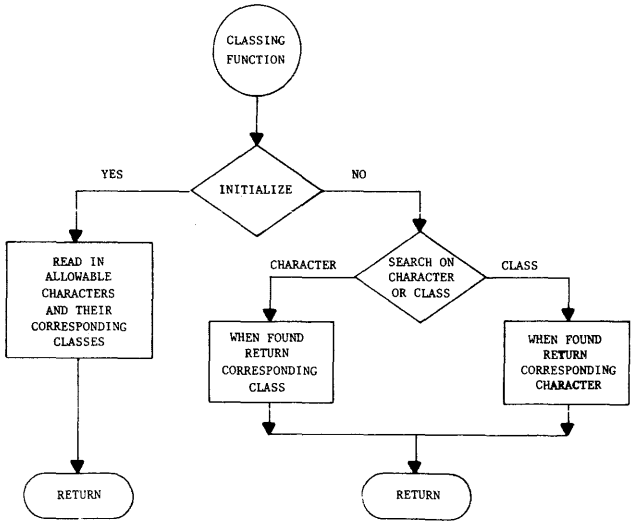


Figure 3 – Classing function

routine, Figure 4, which packs up to three input characters per GE-235 computer word.

Syntax

The syntax routine, Figure 5, effects the syntax checking, controls parsing, and does whatever transformations may be required. The present implementation is a Turing Table[5] and its associated driver.

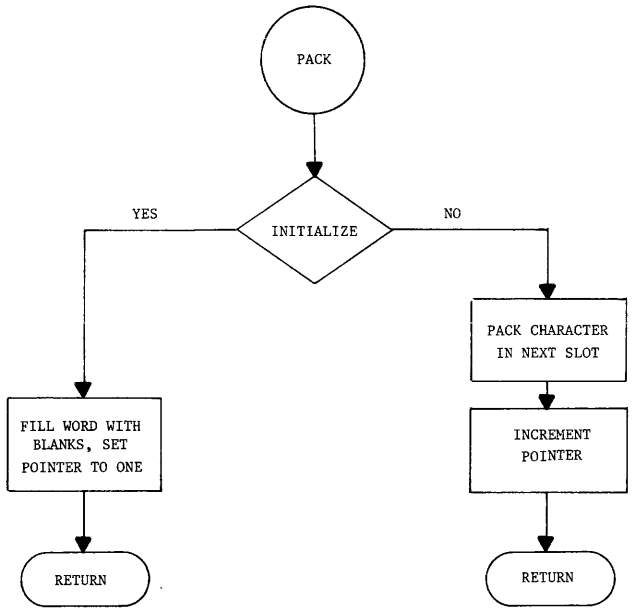


Figure 4 – Pack routine

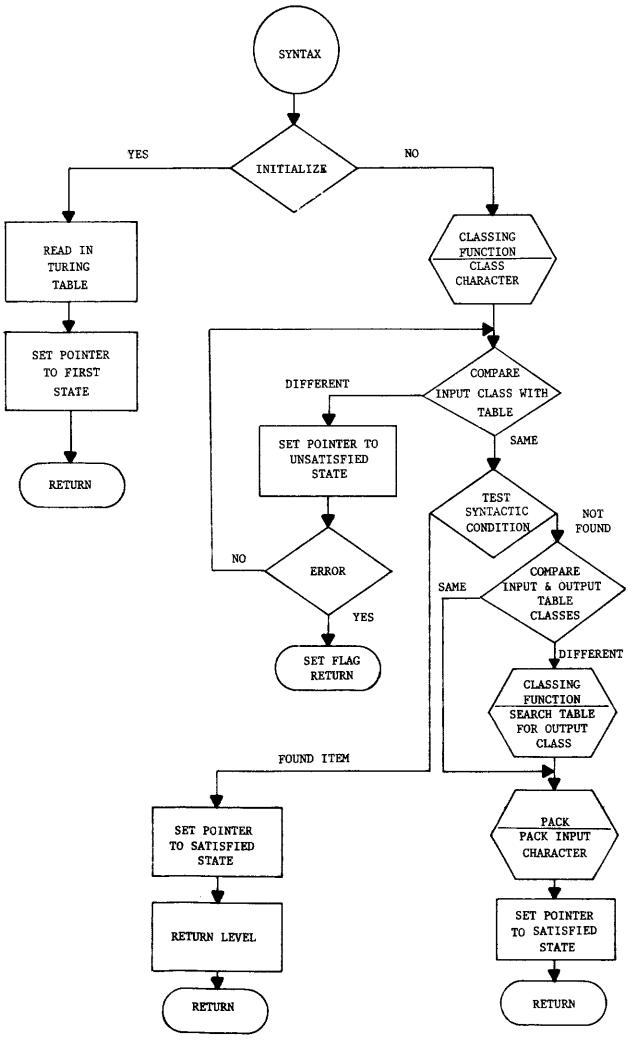


Figure 5 – Syntax analyzer

The table entries are quintuples of the following form (state, symbol, move, new symbol, and new state). The state is determined by position rather than by name. A pointer to the next alternate state is provided to avoid table searches. Moves are presently restricted to "no move" and "move right," with "look-back" being delegated to the input routine. Table self-modification has been provided for, but use of this feature depends upon the meta-language.

The meta-language

As a general-purpose transformation program, one use is the generation of compilers; that is, since the program parses and transforms input on the basis of the syntax table and produces output on the basis of the semantics routines, it can (given a description of a language in some meta-language) produce a Turing Table for that language. This table may then be used with the program to compile the language described. The availability of a satisfactory meta-language greatly affects the utility of the program. Construction of a Turing Table on a line-by-line basis becomes a tedious task if one wished to give the syntactical and transformational rules for a language, like ALGOL or FORTRAN. BNF as a meta-language seems satisfactory for syntactical rules, but will not handle transformational rules. Currently, the program is being used to explore various notations which will combine syntactic and transformational rules as well as allow self-modification of the Turing Table, giving variable syntax and transforms. The ability to bootstrap from one experimental meta-language to another has proved valuable and timesaving, although no entirely satisfactory notation has yet been found.

Symbol table

Once an item is deemed syntactically correct and is ready to be passed to the treeing (parsing) program it first must be logged into the symbol table, Figure 6, which looks to see if that symbol is already present. If it is present, the location is returned; otherwise, it records the symbol at the next available spot, and likewise, returns its location. There is also an entry for returning a symbol given its location.

Parser

The parsing program, Figure 7, manipulates a tree which can be viewed as a two-dimensional stack, with linkages corresponding to the way the items are placed within the tree. It is convenient to link input items into a tree and to remove a node with its corresponding leaves. Also resultant "temp" cells can be allocated and optimized dynamically. Flexibility and control are afforded in a tree structure with regard to what is passed on to the semantics

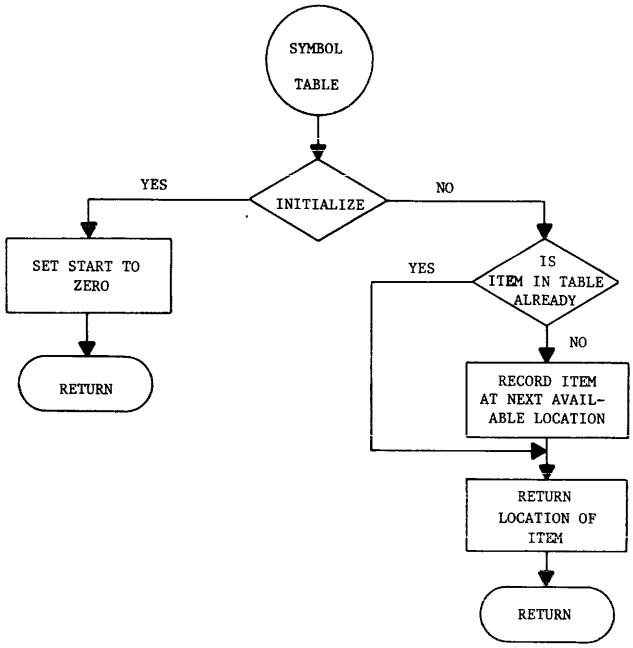


Figure 6 – Symbol table

routines. We use a tree, since much of the literature today uses a tree to depict the syntactic structure of a language. Inherent in the tree structure is considerable power which readily lends itself to such ideas as cross-linked leaves and nodes, and also restructuring of the tree links.

Input to the routine is the location of the syntactic input item from the internal symbol table and its associated hierarchal level. Output is a 4-tuple of the form (node, leaf, leaf, resultant), corresponding to an operator or verb, two operands or objects, and a "temp" result. The mechanics of the routine are simply two programmatical questions which determine where the latest syntactic item should be placed in the tree: Question 1, "Is the new item higher in the hierarchay than the last item in the tree?" If yes, the new item should link up to the last item in the tree; otherwise, Question 2, "Is the new item's level higher than the item to which the last item links?" If yes, swap links; otherwise, either continue asking the above two questions after having first changed the pointer from the last item to its up-link; or pass the last item (which must be a leaf), the item to which the last item is up-linked (which is a node), and any other leaves up-linked to the node, to the semantics routines. The node can be replaced with the location of a temp cell which has been optimized by noting the level of the node and its position.

Semantics routine

The semantics routine receives four arguments which are a representation of the input operator

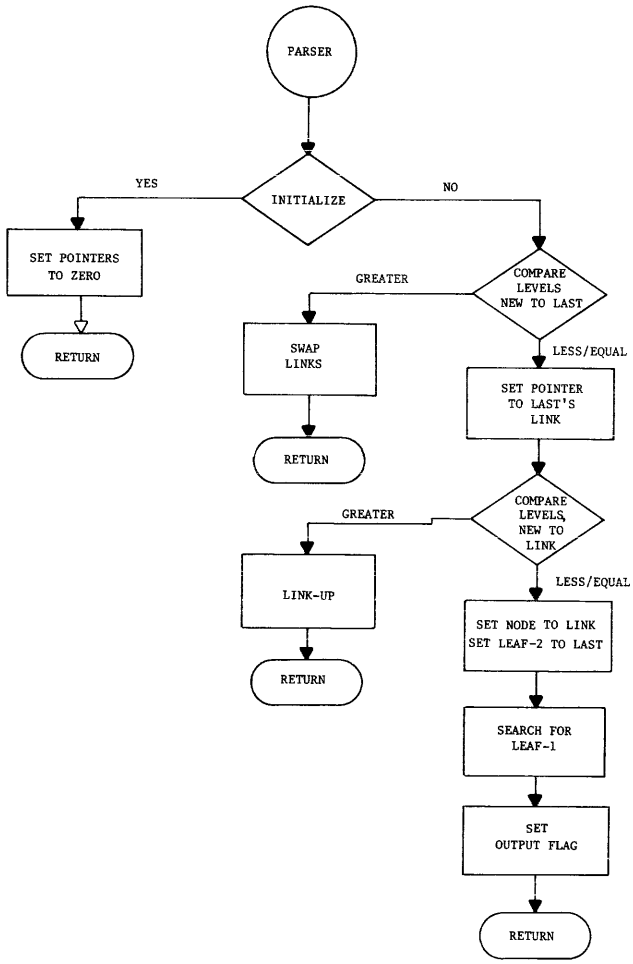


Figure 7 – Parsing function

(verb), two of its associated operands, and a resultant temp cell. At this point the user should have a rough idea of how his input statements will look when parsed so that he will know which of the operands and/or temp cell will be useful for each operator; secondly, he must know what he expects to accomplish in the way of a target language (e.g. FORTRAN, Assembly Code, Turing Table, and so forth). This routine is written by the user for his particular application.

SUMMARY

In conclusion, the program can perhaps best be described as a "skeletal compiler," i.e., a set of routines which function either as a compiler or as a meta-compiler. Syntax checking, character transformation, character grouping, and parsing are under control of a modified Turing Table. The output may be another table in which case the program functions as a meta-compiler. If this new table replaces the original table, the program may be used as a compiler, the output now being assembly code, or calls to functional routines serving as an interpreter.

REFERENCES

1 J E PLASKOW S A SCHUMAN
The Trangen System on the M460 Computer
Massachusetts Computer Associates Inc
Wakefield Massachusetts

2 D V SCHORRE
Meta II A Syntax Oriented Compiler Writing Language
Proceedings 19th National ACM Conference 1964

3 S WARSHALL R M SHAPIRO
A General Prupose Table Driven Compiler
Massachusetts Computer Associates Inc
Wakefield Massachusetts

4 A M TURING
On Computable Numbers with Application to the Entscheidungsproblem
Proceedings of the London Mathematical Society
sec 2 vol 42 pp 230 265 1936 1957

5 M DAVIS
Computability and Unsolvability
McGraw Hill Book Company Inc New York 1958

Architecture for large computer systems

by GERHARD L. HOLLANDER
Hollander Associates
Fullerton, California

Evaluation criteria

As the computer field reaches maturity, users select their equipment not because of internal structure of features, but by its overall *economy*; i.e., performance per unit cost. Here, *performance* means the computer's ability to do the user's job. *Cost* includes not only the purchase price, but also such associated costs as space, power, operation, maintenance, software, and unreliability.[1]

Greater problem complexity stimulates various alternative computer-design approaches; from speeding up conventional processors to using a multiplicity of processors. But if the user is really indifferent to the internal structure provided his job is executed economically, we are confronted with an apparent contradiction. Multiple—and therefore smaller—processors violate the accepted dictum that a larger processor provides more performance per unit cost.[2] According to Grosch's law,[3] manufacturers price equipment to make performance proportional to the square of price; in other words, doubling the price increases the performance fourfold. However, this empirical *pricing* guide stemmed from an era when the manufacturer's only concern was the elasticity of the *total* demand for computers. In a competitive environment, manufacturing cost determines price; and it remains to be seen whether Grosch's law also applies to the *cost* of computers.

In a competitive environment, the user's *economy* becomes the guide of the successful designer. The manufacturing cost of a computer depends on the number and cost of its components. The designer must structure his components so that the fewest and least costly provide the maximum performance. The resultant computer must do the user's job with minimum time expended for overhead or housekeeping operations. Most likely, the components of the most economical computer will have the largest duty factor.[1]

Competitive system structures

Over the last two decades, computer economy has grown one hundred fold. The primary reason is faster circuitry, which provides more performance at the same cost when the faster circuits are in full production. A computer still faster than the largest economical unit available today could be designed by:

1. Speeding up a single instruction stream
   a. Faster single primary execution element (uniprocessor)
   b. Multiple primary execution elements (Array processor)
2. Allowing multiple instruction streams (Multiprocessor)

Uniprocessing:

Most past effort has been devoted to speeding up the primary execution element. Faster circuits and components have been cited already. Another approach is faster units, such as speeding the arithmetic element by parallel instead of serial organization and by high-speed carry. Augmenting units, such as memory look-ahead and separate I/O processors, relieve the primary execution element. Finally a larger primary memory gains time at the expense of storage. All these speed-up techniques have helped make existing conventional uniprocessors a hundred times faster during the last decade.

Array processing

A single instruction stream can be more effective if it can process several data streams simultaneously. For example, a payroll run requires essentially the same operations on the records of hundreds of employees. With an array of a hundred execution elements, the single instruction stream could cause a hundred operations to be performed at the same time.

Two distinct types of array computers have been investigated recently. In one, the associative parallel

processor, each execution element contains little more than the exclusive-or function.[4] Because of its simplicity and low cost, each element can also serve as a storage location; and we might think of all orders being executed in memory (an associative or search memory). Such associative memory costs about one order-of-magnitude more than ordinary core memory, but several orders-of-magnitude less than a complete arithmetic unit.

An alternative array approach employs hundreds of processing elements as complex as a small computer.[5] These elements can execute more complex instructions when stimulated by the primary instruction stream. The processing element's own stored program can be tailored to its specific data stream.

Either form of array processing appears best when the problem itself has geometric properties compatible with the topology of the processing array. When the problem does not have this topological property naturally—for example, inventory or payroll problems—the problem can often be converted to use multiple execution elements effectively.

Array-type processing is inefficient if most processing elements are idle most of the time. In the associative processor, with essentially no decision capability at each processing element, the operations must be identical to the simplest level. The processor arrays, with significant decision-making and logical capability at each execution element, can allow reasonable differences from one execution element to another. However, if most decisions make an element idle, the efficiency will be quite low.

Multiprocessing

The multi-processor computer contains several computers, each capable of executing the job alone.[6,7] They attack a complex problem by segmenting it into many parts and executing each segment autonomously. This allows dynamic allocation of the program among the *available* processors and provides protection against system failure if one of the processors fails.

Multiprocessor-computer task assignments require overhead, either as added hardware or as additional (unproductive) time. Furthermore, if Grosch's law holds also for computer cost, substituting several small processors for a large one starts the trade off in a less favorable performance-per-unit-cost regime.

Comparison of structures

Table I contrasts key characteristics of the four structures.* All but the uniprocessor employ multiple data streams, either as multiple instruction streams with a single data stream, or as single data streams of multiple instruction streams.

Multiple data streams controlled by a single instruction stream must be similar. This restriction is quite severe for the associative processor, because the instruction stream is at the level of Boolean functions. The limitations are strong even for the processor array, where the main instruction stream is closer to a macro-order. While the macro level allows the stored program to adjust each execution element for some data-stream differences, the individual elements must execute the same macro operations or remain idle. For the multiprocessor, the running time of the segments should be within one or two orders-of-magnitude, a minor restriction.

The cost-division values between the portions of the system represent an approximate indication of the processing components for hypothetical "typical" systems. In the uniprocessor, the entire cost is in the central processor. The multiprocessing system has no

*Table I treats pure structures. Some structures in the references are hybrids.

Table I. System-Structure Characteristics

Structure*	Instruction streams	Data streams per Instruction stream	Limitation on data-stream similarity	Approximate cost division				Level of work division
				Central processor	Inter-connection	Other execution elements		
						Total	Each	
Uni-Processor	Single	Single	None	1.0	0.0	0.0	——	None
Multi-Processor	Multiple	Single	Minor	0.0	0.3	0.7	10^{-1}	Sub-routine
Associative Processor	Single	Multiple	Severe	0.6	0.0	0.4	10^{-4}	Boolean
Processor Array	Single	Multiple	Strong	0.2	0.1	0.7	10^{-2}	Macro

central processing component, as each processor is one of the execution elements. However, a significant portion of the cost is in the interconnection and the task assignment. These overhead functions, hardware or execution time, represent approximately 30 per cent.

For the associative processor, the associative memory usually represents less than half of the system cost. For the processor array of the SOLOMON type, most of the cost is in the execution elements. The cost protion of each processing element illustrates the wide range between uniprocessors where the single processing element represents essentially the entire cost, to the associative processors where an element represents on the order of 10^{-5} to 10^{-4} of the cost.

Efficient use of any unconventional processor calls for new programming approaches. Since during the last two decades people have been trained to think serial, at least now the programming for the less conventional systems is more difficult until now procedures are developed. Thirty years ago, would it have been more difficult to train people to think parallel instead of serial?

Which structure is best?

No one structure is best for all possible jobs. The controversy revolves around the relative advantages of each architecture for different applications.

Each of the companion papers[4,5,6,8] attempts to show the superiority of its approach for some or many applications. In fact, Amdahl believes the uniprocessor is best for all but a few special-purpose applications. Table II summarizes each protagonist's view of the appropriate application areas for the different computer structures. Their papers state their reasoning. Unfortunately, or fortunately, an "unbiased" chairman cannot inject his own conclusions into a debate. The four authors speak well for themselves.

Rating Key:
1. Best
2. Acceptable, depending on the application
3. Marginal, depending on the application
4. No reasonable man would consider

Each companion-paper author rated the merit of each structure for the different applications, each large enough to require approximately 10^7 executions per second with a conventional processor. The *rating* columns show the average of their ratings. Unless the value is either near 1 or near 4, it probably does not represent a consensus but an average of divergent opinions. The other column in each group shows how many respondents considered this the best structure for the application class.

REFERENCES

1 G L HOLLANDER
Drum Organization for Strobe Addressing
IRE Transactions on Electronic Computers
Vol EC 10 No 4 December 1961 p 722

2 K E KNIGHT
Changes in Computer Performance
Datamation Vol 12 No 19 September 1966 pp 40-54

3 H R J GROSCH
High Speed Arithmetic
The Digital Computer as a Research
Tool Journal of the Optical Society of America Vol 43

4 R H FULLER
Associative Parallel Processing
AFIPS Conf Proc Vol 30 April 1967

Table II. Consensus of Protagonists

Application	Uniprocessor		Multi-processor		Associative processor		Processor array	
	Ave. rating	No. of bests	Ave. rating	No. of bests	Ave. rating	No. of bests	Ave. rating	No. of bests
Open-Shop Scientific	1+	2	2	1	4	0	4—	0
General Commercial	2	2	2	2	4	0	4—	0
Atomic Research	2—	1	2+	1	3+	0	2+	1
Weather Forecasting	2—	1	2+	1	3+	0	2	2
Pattern Recognition	2+	1	3—	1	3—	1	2+	1
Command & Control	2—	1	2—	2	3+	0	4—	0
On-line Commercial	2—	2	2	2	4	0	4—	0
On-line Scientific	1+	2	2—	2	4	0	4—	0
Info Storage & Retrieval	2—	1	2+	1	2+	2	4—	0

5 D L SLOTNICK
Unconventional Systems
AFIPS Conf Proc Vol 30 April 1967

6 G P WEST
The Best Approach to a Large Computing Capability
AFIPS Conf Proc Vol 30 April 1967

7 G L HOLLANDER
Multiprocessor Systems Organization
IEEE International Convention Record
Convention Record
Vol 13 Part 3 1965 p 213

8 G M AMDAHL
Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities AFIPS Conf Proc Vol 30 April 1967

9 F GRUENBERGER
Are Small Free standing Computers Here to Stay
Datamation Vol 12 No 4 April 1966 pp 67-68

The best approach to a large computing capability

by GEORGE P. WEST
System Development Corporation
Santa Monica, California

INTRODUCTION

The best approach to a large computing capability is to build your own multi-processor system, utilizing the most effective elements available from the leading hardware manufacturers. To understand this approach, it would be useful to describe the type of system which is to be built. It is evident that many of the system characteristics will be prescribed by the application, others will be purely a matter of personal choice. The essential characteristics are few in number and must not be confused with the non-essential characteristics. The remainder of this paper presents a set of do-it-yourself instructions for designing your own multi-processor system which stresses the essential characteristics. The intent being that the reader can follow through these instructions and design his own system for his own particular application. In this way, each reader can have a specific system in mind which he can evaluate critically. Although there would be individual differences in the systems under consideration by different individuals, the essential characteristics would be the same, and discussion would then tend to center about these essential features. The paper concludes with a summary in which the design approach is reviewed in an attempt to emphasize the utility of the approach.

The first step in designing the multi-processor system is to select a computer which is to be the basic module of computational capability. Select one of the many word-oriented full-word computers with a word length of at least 24 bits. Select a computer with multiple memory modules; one with some capability for a shared use of memory. Make the selection based on cost/effectiveness. Select the computer because it gives the most computer power for the money. Don't worry about channel capacity, total memory capacity, etc. These characteristics will be considered later. The criteria for evaluation of competing computers may be simple or complex. If you have no other criterion, you can follow the crowd. Some of the best performing computers in the past have also been the best sellers.

The next step is to select a small character-oriented (byte) computer with good channel characteristics to serve as the I/O, or peripheral, controller and processor. Select the computer so that the large computer's word length is a multiple of the small computer's word length. This is relatively easy for a character-oriented small machine. We will decide how many of each type of module to provide later.

Now, since we have selected a small general-purpose computer as the I/O controller, we will shop for peripherals that have a minimum of control logic in each unit. Simply look for good clean electro-mechanical design and select each peripheral from the most appropriate vendor. It is very unlikely that any one vendor can supply more than one or two types of modules for the system.

Estimate the total system computational capacity, based upon the particular requirements of your application; then select the number of large processors based upon this computational load. Do not include extra capacity for executive functions. The number of memory modules to be provided is based upon the performance you wish to provide to the different classes of users. In general, two memory modules per processor will provide overlapped swap and computer capabilities for a large class of applications. A relatively simple computation will provide estimates of the system performance with different numbers of extra (swap) memory modules. The computation will reflect your own mix of problems, the grade of service you hope to give each class of user, and the expected loading for each class of user. These performance estimates can be as simple as statistical estimates derived from drawing samples from a set of hats with the correct mix of alternatives in each hat, or as sophisticated as detailed simulations. Results derived from published queueing tables

should provide sufficient accuracy for determination of the size of the system memory. Next, determine the number of peripherals required for your application, and from this estimate how many small peripheral processors will be required to support your I/O functions. In addition, these small processors must also perform some executive functions, which should require no more than about ¾ of the capacity of one of the small computers. This estimate depends upon your own particular application as well as upon the amount of I/O supported. If there are several I/O processors, the executive can be interleaved with I/O idle time so that little extra capacity is required. If your I/O requires most of one small computer's capacity, the executive function might necessitate a separate processor.

The system components must now be interconnected by designing a high-speed communications network so that the memory modules are all accessible to all computers. This network may take the form of a multiple-path, high-speed switch between the processors and the memory modules. It should include special logic to facilitate the executive function. In particular, it should provide hardware for translation of symbolic memory addresses into actual cross-point locations on the switch. It should include hardware to facilitate queueing of processor assignments and should provide for interrupt routing to the appropriate processor.

In addition, a low-speed network should be provided for connecting the peripheral units to the I/O processors. This network might take the form of a multiple-path relay switch. It requires only very simple control logic since it will be controlled by the I/O processors, which execute only I/O and executive programs. Peripheral reconfiguration will seldom occur. I/O functions would normally be grouped, and assigned in groups to each of the I/O processors. In addition to the assigned I/O functions, any of the I/O processors could be preempted to perform an executive function, such as swapping a program segment in core with one on the backup storage (probably drum or disk). This low-speed network should be required only if graceful degradation is desired.

The final step is to write the executive program and I/O programs for the small I/O processor.

The full-word processors are to be used only for execution of object programs. In order to do this, we must consider the type of object programs which the users of the system will provide. Let us, therefore, define certain aspects of the object programs which are required by this system. The entire computational task, from initial input to final output, in support of a given user of the system is defined as a job. When each task is initiated, it is assigned a sequential job number by the system. If a user attempts to initialize a task which is already in execution in the system, the job number will permit the system to distinguish each use as a separate task. The job number is utilized to assure privacy between users, in that different jobs are not permitted to interact. Communication between jobs is permitted through use of system tables specifically designed for common use by two or more jobs. The system I/O routines also permit shared use of a peripheral storage medium, such as magnetic tape or disk packs, for communication between jobs. Each job is to be broken down into threads of computation. The maximum length of each thread is set by the core limitations of the memory module and the maximum execution time imposed by the executive system, so that, if the time limitation is exceeded, the job may be temporarily swapped out of core.

Threads of code are linked by a macro instruction CUT to X which initiates subsequent threads within the job. The system does not utilize re-entrant code. That is, only one processor will execute a thread of code at a time. System tables and subroutines are replicated in the environments of different threads of code. When a large computer and a memory module are associated in computation, there will be very few memory cycles available for other uses without slowing down the computations. For this reason, system tables can be designated as guaranteed, in which cases the tables will be restored to the system data base when the task is completed, thereby, updating automatically, the contents of the table in the system data base. System macros provided permit the program to designate parallel threads which could be executed concurrently. The programmer may use recursive coding techniques so that parallel sections of code may be nested. The executive system which includes both hardware and software, does not recognize a macro instruction for automatically synchronizing the tying back of parallel paths. This bookkeeping must be provided in the object code itself.

There is no limitation to the user of the system if the synchronizing code is generated by the compiler rather than by requiring the executive program to effect the synchronization. By limiting the executive functions to essentials, the basic system approach may be more easily understood. For this discussion, the non-essential software features, such as the compiler and diagnostic aids are not considered part of the system. Without carefully defining these non-essential software features, there would be a great deal of un-

certainty as to the overall cost of the software system.

There are three system macros provided for programmer identification of parallel threads of computation. They are:

Cut — Enter a memory assignment in the queue waiting for a full-word processor to become available.

Wait — Release full-word processor for a new assignment, but retain this thread of computation and its environment (subroutines, tables, etc.) in core, if at all possible.

Release — Release full-word processor for a new assignment and release the program thread so that guaranteed data items can be updated in the system data base, and a new program thread can be loaded by the I/O processors.

In addition to these system macros, there are three system queues: one for the full-word processors and the other two for the small I/O processors. The queues are:

Full-Word Processor Queue — Computational threads which are loaded and ready for execution.

Executive Queue — Calls for executive functions, primarily swapping of program threads between drum (or disk) and core.

I/O Queue — Calls for I/O. This includes routine loading of program threads upon release of a memory module.

Both I/O and Executive calls are handled by the small I/O processors. However, the Executive calls take precedence over the I/O calls. These queues would be implemented in the high-speed communications network so that, under normal operating conditions, all processors would simply accept an assignment from the queue and proceed as far as possible with that assignment before accepting the next assignment from the queue. Priority logic complicates this simple mode of operation only by use of the interrupt feature to preempt certain computational assignments. The interrupt releases the processor and automatically inserts a cut into the appropriate queue to reinitiate the interrupted task at a later time.

With this design philosophy, the executive and I/O functions can be developed for the small I/O processors. The system design is now complete.

Without specific applications in mind, cost estimates would be meaningless. The system has been described in such a way as to identify essential functions while simplifying interface considerations so that each element of the system may be costed independently. Having completed the design for your particular application, you should now be able to estimate the total system cost. For a large class of applications, this approach provides the best way of attaining large computational capability.

The system outlined has several interesting characteristics.

1. It provides a high-performance system since we have delineated specific capabilities, such as arithmetic, memory, and I/O, and optimized each independently from a cost/effectiveness point of view.
2. System overhead time is minimal since the central computers perform only job computations. The executive time is confined to a partial use of the character-oriented I/O processors.
3. System overhead hardware costs consisting of the high- and low-speed interconnection networks are clearly delineated.

I/O and Executive software development costs contribute to system overhead. If more manufacturers followed the practice of providing separate cost for hardware and software, a portion of the software costs would be offset by the hidden cost of whatever software operating system the manufacturer provided with his hardware. However, by applying currently emerging management techniques to software development, the system software can be produced at a lower cost

Let us summarize by attempting to review the rationale behind this particular design philosophy. The use of peripherals with minimal control logic selected from many vendors will provide the least expensive user interface if the control functions stripped from the peripherals can be supplied at the same cost by the data system. The use of character-oriented small computers instead of special I/O controllers provides an improved level of support to the peripherals and their users at a cost lower than that of special-purpose controllers. The user's arithmetic support is provided by selecting the most economical word-oriented computer for that task. The central question then is whether the high-speed switch and its associated queues simplify the executive functions sufficiently to warrant the extra hardware costs.

Associative parallel processing

by R. H. FULLER
General Precision Inc.
Glendale, California

INTRODUCTION

Several previous investigations[1-5] have shown associative processors to have utility in solution of problems which allow the same operation to be performed simultaneously over many elements of a data set. Examples of such problems include picture processing, matrix manipulation, signal correlation, ELINT data processing and information retrieval. It is the purpose of this paper to contrast the efficiency of solution of such problems on an associative parallel processor to efficiencies obtained using other computer organizations described in this session. Efficiency is measured as solution rate per unit cost.

The associative parallel processor (APP)[5] is an array computer exhibiting parallelism of instruction execution similar to that obtained in SOLOMON type machines. It differs from SOLOMON machines in having much less storage (e.g., a single associative memory word) dedicated to each array element. Logic hardware at each array element is minimized by use of a novel computing technique termed Sequential State Transformation. By this technique, data stored within an associative memory having a multiwrite capability may be transformed according to any Boolean function of stored and external variables.

Cell cost for APP is two to three orders of magnitude less than for a SOLOMON type cell. Instruction execution times for APP cells are longer than for SOLOMON cells by one to four orders of magnitude, dependent on instruction complexity. An APP should be more efficient than SOLOMON type machines in applications having small cell memory requirements and simple data manipulation at each cell. Such applications include picture processing, information retrieval, signal correlation and simple matrix manipulations (e.g., solution of assignment matrices). Solutions of complex field problems, particularly when variable ranges dictate floating point number representation, are not normally efficient on APP, even though these problems allow parallel computation requisite to efficient use of an array computer. Problem solution time for an APP can only approach that of a SOLOMON array if the APP contains, and may effectively use, many more cells than the SOLOMON array. An APP would typically contain several thousand cells as opposed to several hundred for a SOLOMON array.

In regard to generality of the various processors discussed in this session, the conventional uni-processor employs a single instruction stream operating on a single data stream and is thus uniformly effective on all problems capable of digital solution. An aggregate of conventional uni-processors employs multiple independent instruction streams, each operating on a single data stream. This organization is effective whenever a problem can be partitioned into several subproblems, each of which allows of independent solution. An aggregate of uni-processors is less general than a single uni-processor in that problems which do not allow partitioning do not effectively utilize the machine.

An array processor employs a single instruction stream operating simultaneously on many data streams. This mode of operation can be achieved with an aggregate of conventional uni-processors by supplying the same instruction sequence to each uni-processor. An array processor is thus less general in application than an aggregate of uni-processors.

Among array processors, an APP is further restricted in application than a SOLOMON type processor, in that floating point arithmetic operations are not available within APP. An APP thus has restricted utility in some scientific computations for which a SOLOMON machine is well suited.

It is felt that within its useful range of application, the efficiency of APP exceeds that of other organizations discussed in this session. To illustrate this point, we shall, in the following section, review the organization of APP and discuss its utility in picture processing, a task to which the APP is well suited.

Processor organization and command set

To illustrate the concept of sequential-state-transformation, consider an associative memory which stores two operands, A_i and B_i, in each work of memory. We desire to add operand A_i to operand B_i simultaneously in some subset of these words. Processing is serial by bit, and parallel by word, starting at the least significant bit of each field. Each word has an auxiliary storage bit, C_i stored within the memory array. Bits within operand field A are designated A_{ij} (j = 1, 2, . . . , N), where N is the field length. Bits in field B_i are similarly designated. The truth table defining the addition is as follows:

State number	*Present state*			*Next state*	
	A_{ij}	B_{ij}	C_i	B_{ij}	C_i
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	1

Note that variables B_{ij} and C_i differ in their present and next states only in states numbered 2, 4, 5, 7. The search and multiwrite operations may be used to perform the addition of all number pairs, starting at the least significant bit, as follows:

1. Search words having $A_{ij} = 1$, $B_{ij} = 1$ and $C_i = 0$. For these words, multiwrite $B_{ij} = 0$, $C_i = 1$.
2. Search words having $A_{ij} = 0$, $B_{ij} = 0$ and $C_i = 1$. For these words, multiwrite $B_{ij} = 1$ and $C_i = 0$.
3. Search words having $A_{ij} = 0$, $B_{ij} = 1$ and $C_i = 1$. For these words, multiwrite $B_{ij} = 0$.
4. Search words having $A_{ij} = 1$, $B_{ij} = 0$ and $C_i = 0$. For these words, multiwrite $B_{ij} = 1$.

Steps (1) through (4) are repeated at each bit of the operands. Within each bit time, processing is sequential by state over present states which differ from the next state in one or more variables. All words in a given present state are transformed simultaneously to the desired next state.

Sequential-state-transformation, used to perform the above word-parallel, bit-serial addition, is evidently a very general mode of associative processing. It allows transformation of memory contents according to any Boolean function of stored and external variables. It makes full use of comparison logic, implemented at the bit level within an associative array, and thereby simplifies logic required at the word level.

In the following we describe the organization and command set for a processor using the sequential-state-transformation mode of associative processing.

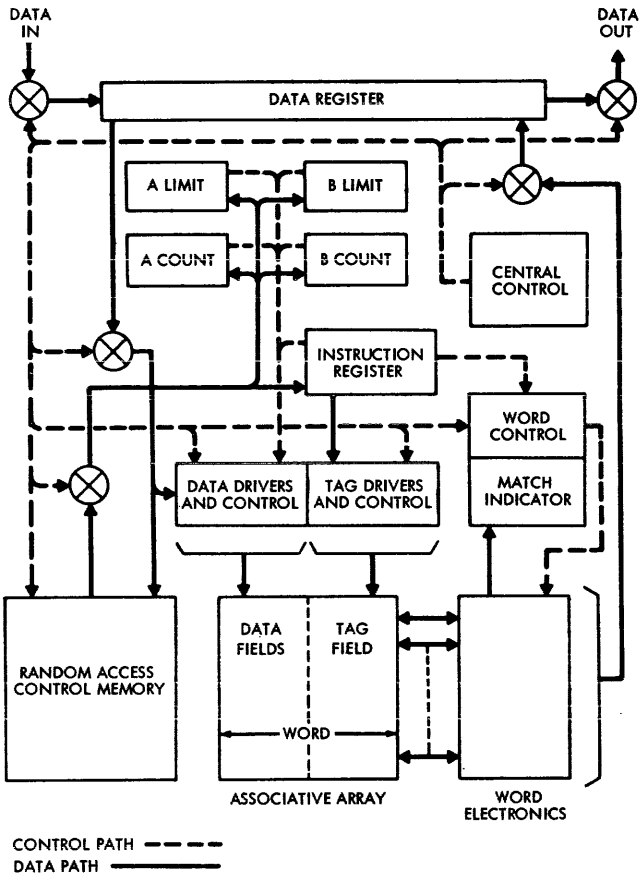


Figure 1—Structure of the associative parallel processors

Elements of an associative processor are shown in Figure 1. The format for associative commands is shown in Figure 2. Each associative command effects a primitive transformation of state variables as discussed above. The left-most bit identifies the command as associative. The two adjacent bits define the initial state of match flip-flops in word logic units (i.e., the detector plate). Other bits define search and rewrite criteria for the A field, the B field, and for each of four tag bits. The right-most bit controls rewrite into matching words or their next lower neighbors. Functions of these bits are described in Figure 2.

To illustrate the utility of this command, consider the task of searching the associative memory for words matching the data register over a field having its upper limit stored in the A limit register and its lower limit stored in the A counter. Matching words are to be tagged in tag bit 1.

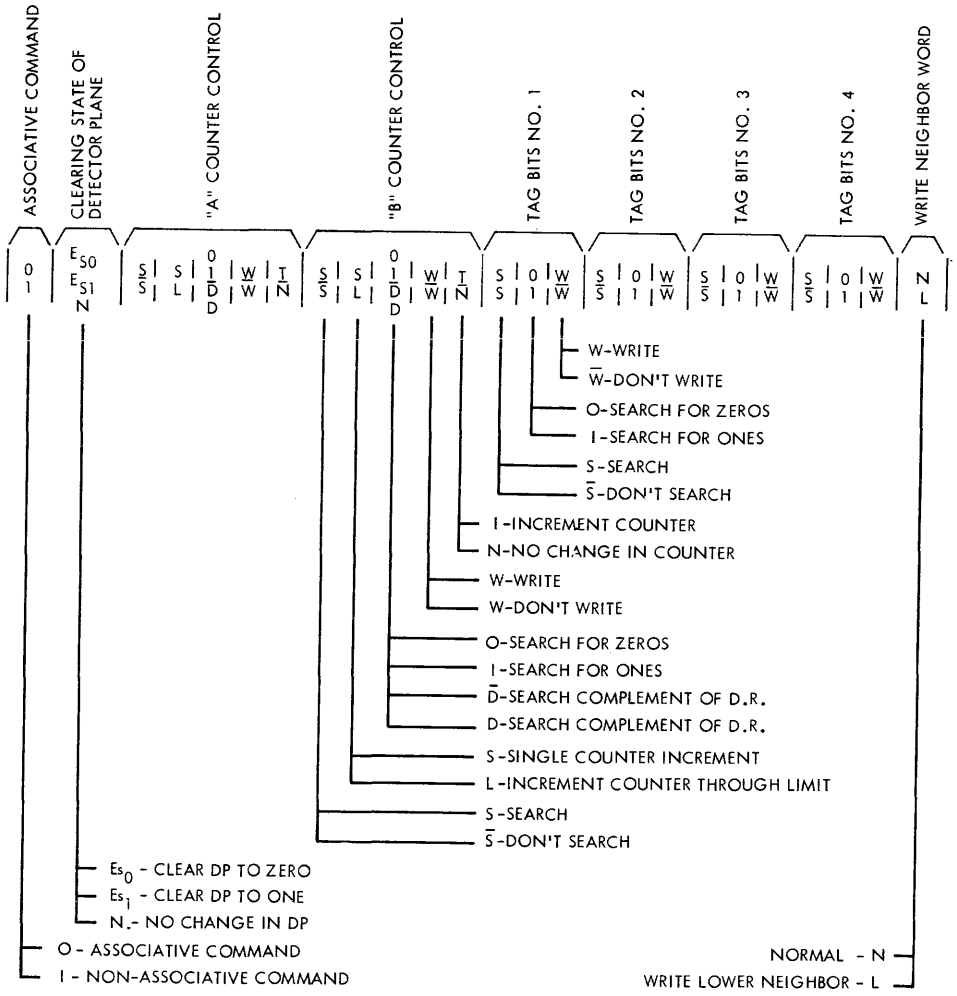


Figure 2—Format for associative command

The following command accomplishes the desired tasks:

1 E_{si} S L D W I S — — W — S O W
A control B Control Tag 1

The following routine loads data into each word in the associative array. The word field to be written is again defined by contents of the A counter and the A limit register:

1. Set the match flip flop for word 0 to "1."
2. 1 N S L D W S — — — — S — W N
3. 1 N S L D W S — — — — S — W L
   A Control B Control Tag 1

4. If no match, exit: otherwise go to (3).

Instruction (2) writes into word 0; instruction (3) writes sequentially into each remaining associative word.

Nonassociative commands are provided to load the A and B counters and limit registers, to branch from linear instruction sequencing either unconditionally or when speciifed conditions are met, and to input or output data.

Associative pattern processing

The parallel processing capability of an associative processor is well suited to the tasks of abstracting pattern properties and of pattern classification by linear threshold techniques.[6-8] Threshold pattern recognition devices execute a given operation independently over many data sets, and thus allow the parallelism necessary for efficient associative processing. Associative processing affords the accuracy of digital number representation, and is thus unlimited in fan-in and dynamic range of weights. Weights are simply altered by changing memory contents. Wiring and components are regular and are thus amenable to low-cost, batch-fabrication techniques. The set of measured pattern properties is changeable by changing memory contents, rather than by rewiring as for analog units. Adaptation is thus possible in measured properties as well as in classification.

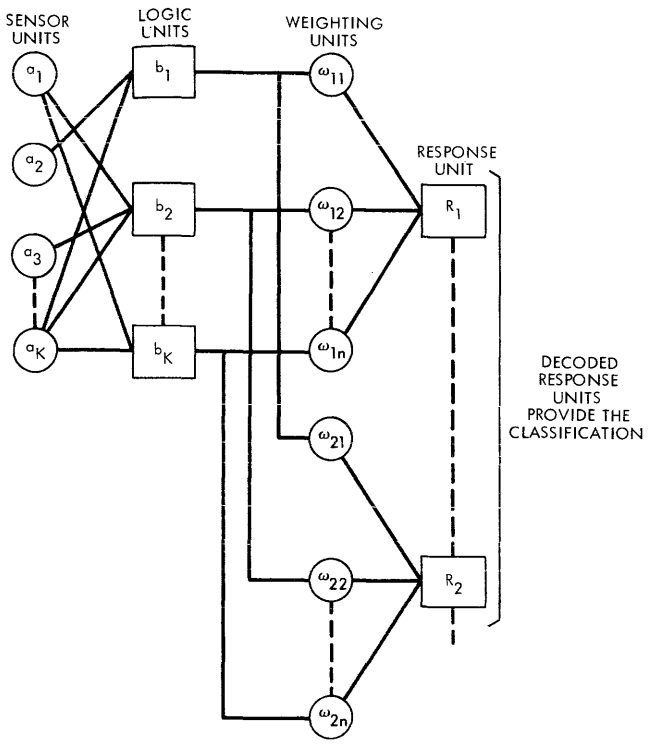


Figure 3—Analog model of pattern recognition system

Figure 3 represents the model of the pattern recognition system to be realized. Binary valued sensor outputs are summed, with weights ± 1, into some or all of the thresholding logic units. A threshold level is established for each logic unit. If the sum of binary weighted inputs exceeds the threshold, the unit becomes active and its output is "one"; otherwise the output is "zero." Each logic unit has a weighted connection to some or all of the response units. Weights of active logic units are summed and thresholded at each response unit. A pattern is classified according to the set of activated response units.

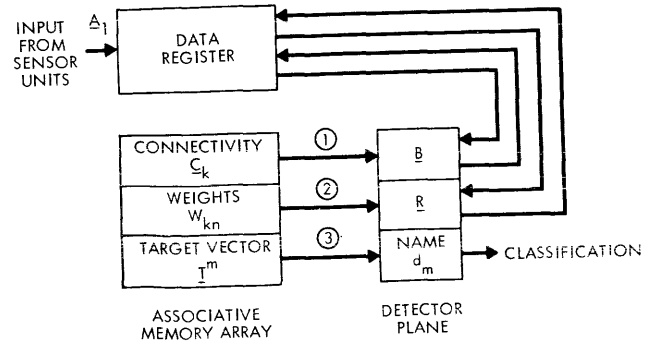


Figure 4—Associative parallel processor realization of pattern recognition system

To realize this model, the associative memory is organized into three sections containing, respectively, a connectivity matrix defining connections between sensors and logic units; the system of weights associated with inputs to response units, and the target vectors associated with patterns to be recognized. The general organization of the associative memory is shown in Figure 4. Processing takes place as follows: In Phase (1), the set of logic units activated by the input pattern is determined, using the input pattern and the stored connectivity matrix. Logic unit outputs are formed in the detector plane. In Phase (2), the inputs to the response units are calculated, using logic unit outputs and the weights stored in the second sector of the associative memory. This yields the response unit outputs in the detector plane. In Phase (3), the response unit outputs are compared with the target vectors stored in the third sector associative memory, and the pattern classification is determined.

Figure 5 illustrates pattern recognition times for an associative processor having an execution time for associative commands of 0.8 microseconds. Here, "N"

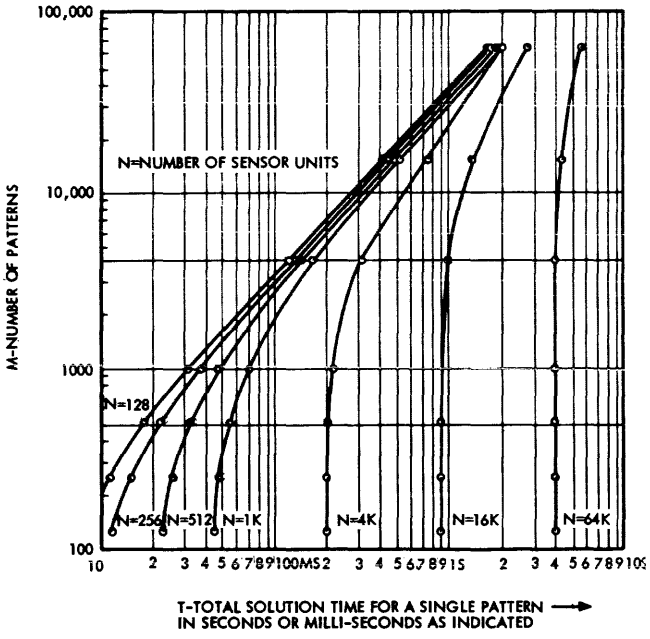


Figure 5—Time for associative classification of a single pattern as a function of the number of patterns and sensors for the W/C data organization

represents the number of sensor units at the input and "M" the number of patterns distinguishable by the processor. The APP can classify an input pattern employing 400 sensors and 512 logic units in approximately 3 milliseconds. Some 2000 words of associative storage are required.

CONCLUSION

The associative parallel processor, described in this paper, achieves considerable generality with simple word and bit logic through the use of the sequential-state-transformation mode of associative processing. An important feature of the parallel processor, when used as a pattern recognition device, is the ability to modify its functional structure, through alteration of memory contents, without change in its periodic physical structure. This adaptive feature has importance in applications where patterns change with time, or where the processor is used as a prototype of subsequent machines having fixed recognition capabilities.

Linear threshold pattern classifiers of the type here presented are beginning to find many applications. To date, these types of pattern classifiers have been studied and/or implemented for character recognition, photointerpretation, weather forecasting by cloud pattern recognition, speech recognition, adaptive control systems and more recently, for medical diagnosis from cardiographic data. Other possible applications include terminal guidance for missiles and space vehicles and bomb damage assessment.

In addition to the pattern recognition task herein described, the APP has also been applied to the tasks of job shop scheduling, optimum commodity routing and processing electromagnetic intelligence (ELINT) data. In each instance significant speed gains have been shown possible over conventional sequential digital computers. It is interesting to note that the processor described in the second section of this paper may be applied to this variety of tasks without significant changes in organization or command structure.

REFERENCES

1 G. ESTRIN and R. H. FULLER *Algorithms for content-addressable memory organizations* Proc. Pacific Computer Conference pp. 118-130 March 1963

2 Ibid. *Somé applications for content-addressable memories* Proc. Fall Joint Computer Conference pp. 495-508 Nov. 1963

3 P. M. DAVIES *Design of an associative computer* Proc. Pacific Computer Conference pp. 109-117 Mar. 1963

4 R. G. EWING and P. M. DAVIES *An associative processor* Proc. Fall Joint Computer Conference pp. 147-158 1964

5 R. H. FULLER and R. M. BIRD *An associative parallel processor with application to picture processing* Proc. Fall Joint Computer Conference Nov. 1965

6 R. WIDROW et al *Practical applications for adaptive data processing systems* 1963 WESCON Aug. 1963

7 C. H. MAY *Adaptive threshold logic* Rept. SEL 63-027 (TR 1557-1) Stanford Electronics Lab, Stanford, Calif. Apr. 1963

8 G. NAGY *System and circuit designs for the tobermory perceptron* OTS AD 604 450 Sept. 1963

Unconventional systems*

by DANIEL L. SLOTNICK
Department of Computer Science
University of Illinois
Urbana, Illinois 61803

INTRODUCTION

Background observations

For the bulk of the past decade, the computer manufacturer* has maintained the "party line" that their successive models of high-end machines possessed all the performance that would be required for the foreseeable future. (This "party line" alternated with abortive attempts to, in fact, build high performance, compatible extensions of the product line which turned out to be neither high performance nor compatible.) While it appears to be true that currently available equipment will for a while continue to be adequate to permit plumbing supply houses to do inventory control, it has always been outrageously false that this equipment come evens close to meeting our scientific or military needs. In these areas, we have witnessed the oft noted paradox of technological advances spurring scientific needs in such a fashion that the disparity between what is needed and what is available continues to grow. In the case at hand, this is no paradox at all in that it is only now that we can realistically envision computers of sufficient speed and capability to make it possible to start using them to solve problems with a non-trivial intellectual content.

Two related sequences of events have in very recent years made it no longer necessary to apologize for grown men to be concerned with producing computing machines of vastly greater capability. These are the consequential gains made by a technology (no thanks whatever to the manufacturer) that was not obliging enough to stand still and secondly that we have the possiblity of performing for the first time with the latest generation of equipment, calculations that couple closely to the mental and physical processes of scientific advancements.

Current status review

A startling result of the past decade of progress in switching elements is the number of such elements which can be employed in a system of given reliability. For example: A system which contains 10 to 20 thousand vacuum tubes achieved mean-time between failures of at most several tens of hours. Comparable or even somewhat greater mean-time between failures are achieved today by systems with upwards of 10^6 conventional transistors. Within the next five years with the coming large scale integration, this number should go to between 10^8 and 10^9 discrete transistor equivalents.

Now, granting the assumption that we will in the foreseeable future be limited by computer capacity in a broad spectrum of areas, it is legitimate to ask that organizational concept be evolved which can utilize a greater number of components to achieve at least proportionally greater performance. It is an important practical aside to note that during the same interval of time the cost of a small signal switching element has gone down to an extent permitting the employment of the maximum number which in non-redundant designs yield a reliable system.

Another important factor affecting our approach to computer design is the growing trend to employ design mechanization. Although this field is still very young it has already advanced sufficiently for us to forecast confidently that reduced development costs and lead times should continue to make successively bolder progress tolerable. I am by no means suggesting that computers of vastly greater power will be designed more cheaply and more quickly than their predecessors, but that as a percentage of our gross national product, these costs will be tolerable in their relative magnitude and continue to yield the nation a reasonable return.

*This work was supported in part by the Department of Computer Science, University of Illinois, Urbana, Illinois, and in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144.

*The term manufacturer is used here only to describe large manufacturers, say those possessing more than half of the total computer market.

The manufacturer continues to bombard our (mercifully) somewhat desensitized ears with the problem of software conversion in an effort to perpetuate hardware that when looked at even ten years from now, will have value only to the historian or antique collector. (In the case of recent compatible families this argument has changed somewhat in that the manufacturer finds that offering even paltry improvements in equipment eliminates compatibility. The new argument is that only the manufacturer can produce software in a timely, economical fashion because of his ability to pool requirements from different customers. The supreme irony of this point of view is already legion.) The value of the machine software investment (notice I said value not cost) has been greatly exaggerated for the same reason by the manufacturer. In particular, the sophisticated user (i.e., the academic and military eggheads with their huge problems) have traditionally been required to program their problems in a low level language. The argument that we must make no further progress in computer organization—a field that is no more than 20 years old in order to maintain a fictitious compatibility with Baggage's computer and its electronic progeny is arrogant and patently absurd.

In seeking up-to-date criteria for judging the efficiency of organizational concept the following are obviously suggested:

(1) That they make efficient use of high-level semiconductor components (i.e., equivalent to from several hundred to several thousand discrete elements), and

(2) That the designs keep a maximum number of memory amplifiers working, i.e., maintain maximal traffic on memory distribution busses both productively and without chaos.

A third criterion is the extent to which a given design can employ design mechanization in its implementation. An extention of this last criterion threatens at some time in the future to explode all our conceptions about computing machines. It is already possible to conceive a machine which will automatically design and fabricate special purpose computers on the basis of a set of instructions not differing greatly from a current machine program. In other words, the filing cabinets full of punched cards would be replaced by filing cabinets full of special purpose computers. It is my current plan to pursue such machines when the current machine we are building at the University of Illinois (ILLIAC IV) is successfully operating. I am greatly heartened that some of the same people who five years ago reacted to the idea of ILLIAC IV with smug incredulity are reacting now in the very same way to this idea.

Some approaches to large computing capability

To return to 1967: What are the approaches currently being explored to achieve greater speed? The manufacturer has a simple answer. If a problem requires a 1000 times the power of computer "X," use a 1000 computer "X's." Of course, this ignores the fact that attempts to achieve cooperative interactions between general purpose computers using clip leads and other strong medicine have in the past yielded lamentable and sometimes laughable results. This approach we can dismiss out of hand if for no other reason than that it is ponderously inelegant; but there is another reason—it doesn't work.

Another approach is to simply state that the conventionally organized general purpose computer will continue to improve so as to keep pace with requirements. This argument ignores the fact that the requirements which are not generated by the existence of a given computer at a given place, for example those motivated by scientific or military problems where the outside world determines the problems, have already produced requirements greatly in excess of current capabilities. Secondly, as has been pointed out now for many years, improvements in general purposes machines are not open-ended because of speed of light and memory access and buss distribution limitations.

Next, in the current set of ideas to achieve high performance are a number of concepts which achieve their very high performance partially by sacrificing some generality of purpose. I must remark here about the erroneous impression created by referring to our current machines by the name "general purpose" computer. This name makes the false implication that they are uniformly good (or poor) at all things. Moreover, the only high performance computer* currently in operation differs markedly in its organization from its predecessors and moreover experience with this machine seems to indicate that the variation in its efficiency of application from problem to problem is quite considerable.

The "pipe line" systems to be available in the next several years offer another order of magnitude of performance over the fastest machine currently available.

These "pipe line" machines achieve speed through the same basic mechanism as parallel machines, i.e., applying a single instruction to a considerable block of data. They, as a consequence, suffer from the same general disadvantage, namely: If the problem does not permit the data to be treated in blocks, their efficiency degrades. Most problems examined, how-

*This is of course, the Control Data 6600 soon to be succeeded by a related machine which is four times faster.

ever, that require very high performance do seem to possess this property of permitting data to be handled in blocks to a remarkable degree although this is not always obvious at first glance. The "pipe line" approach is novel and merits our closest attention. It is being pursued by the major manufacturers in the field.

The approach that I am personally pursuing is the parallel approach. It is the only current approach which is open-ended. The same remark holds for both parallel and "pipe line" systems, namely: Its generality is not fully established.

I will furnish a brief overall description of the system here and refer you for details to a separate session at this conference dealing exclusively with the ILLIAC IV hardware, software and applications.

The ILLIAC IV system

Simply to illustrate the capacity which current technology permits in a computing system, I will give a brief description of the ILLIAC IV system being developed jointly by the University of Illinois and industry.

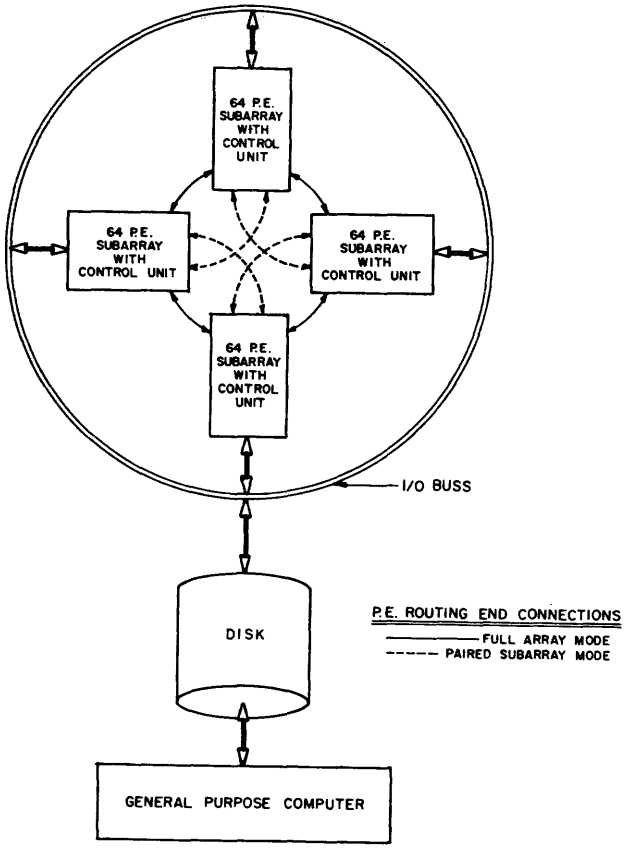


Figure 1 – ILLIAC IV General Organization

Figure 1 illustrates the general arrangement of the ILLIAC IV system, centered on four subarrays, each containing 64 processing elements under the direct control of a subarray (Figure 2) control unit.

Data is transferred between the memory units of the processing elements of the subarrays and a large scale disk file buffer memory via a highly parallel input/output buss. Such input/output transfers are controlled by an external general purpose computer which also supervises the ILLIAC IV program runs. The general purpose computer is provided with a limited set of connections to the subarray control units for this purpose.

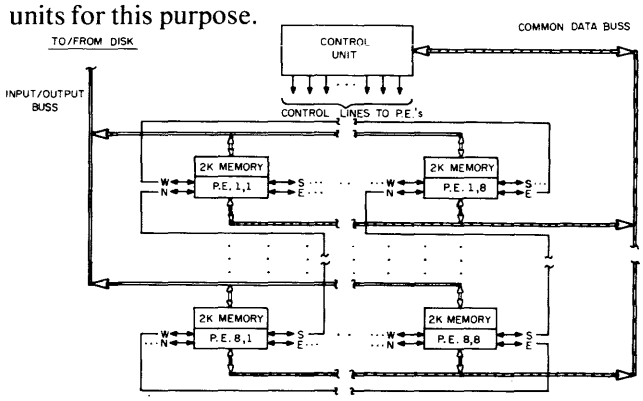


Figure 2 – Organization of 64 P.E. Subarray

The routing connections of the processing elements in the four subarrays are arranged to permit a *united mode* of operation, in which the four subarrays act as a single large array for routing purposes, a *paired subarray mode* of operation which provides routing for two arrays, each containing two of the subarrays, or an *uncoupled mode* in which each subarray operates independently.

Figure 3 shows the structure of one Processing Element (PE). Each PE is provided with three 64-bit arithmetic registers and high speed adders for full 64-bit floating and fixed point operations. The processor is also provided with 2000 64-bit words of thin film memory.

Four routing connections, identified as North, East, South and West are provided in the subarray as shown in the figure together with busses to and from the control unit and the disk unit for global and input/output data.

It is a vital consideration that the PE is regarded as a complement. It is, in fact, delivered in assembled, tested form to the system supplier by the semiconductor manufacturer. Provision is made in the design of the PE to permit an orderly transition from hybrid LSI components to monolithic LSI components during the course of the program. The technology employed in the design and fabrication of the PE will, it is anticipated, yield a unit price for the first system of this very powerful 64-bit floating point unit of under $10,000.

Detailed analytical and programming work has been done in the following areas:

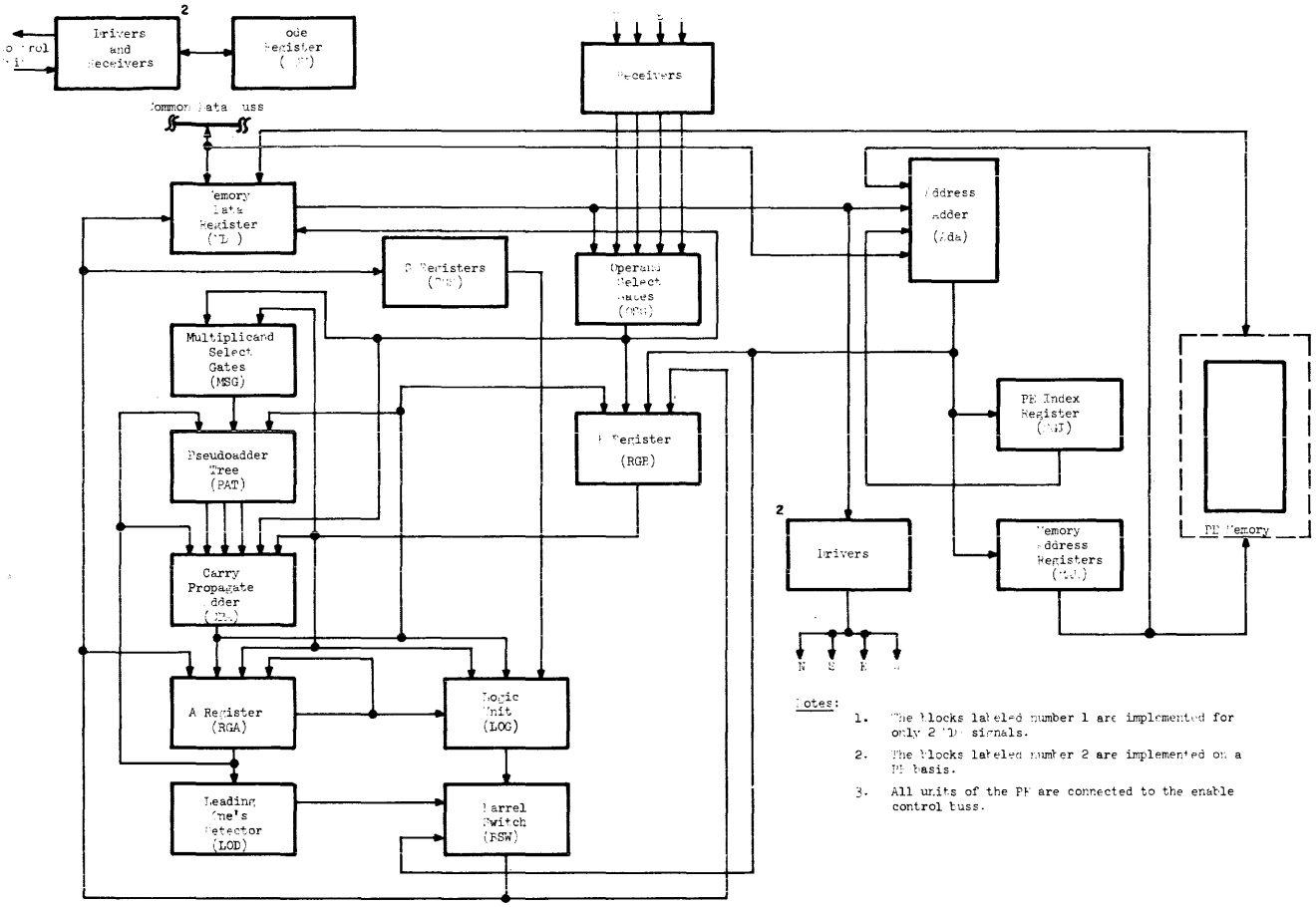


Figure 3 – Processing Element Component, Block Diagram

(1) Weapons Effects
   a. Eulerian flow
   a. Eulerian flow
   b. Lagrangian flow
   c. Neutron transport
   d. Underground stress-strain model
(2) Alternating Direction Implicit Relaxation
(3) General Circulation Weather Model
(4) Matrix Operations
   a. Inversion
   b. Eigenvalue calculation
(5) Linear Programming
(6) Multichannel Filter Design and Convolution
(7) Fourier Transformation

In all the areas listed it was found that highly efficient methods could be evolved, i.e., methods which keep almost all PE's running almost all the time. Thus, the speed-ups observed have been in the order of 256 times N where N relates the speed of the PE to the speed of the computer selected for comparison.

As an illustration of the sort of consideration involved in securing high efficiency on a parallel machine, Figure 4 shows the memory allocation for a 5×5 matrix in 5 PE memories (PEM's). We refer to this storage scheme for matrix elements as skewed storage.

The main reason for skewing a matrix is that rows and columns of a matrix can be accessed with appropriate PE indexing. For instance, to fetch elements

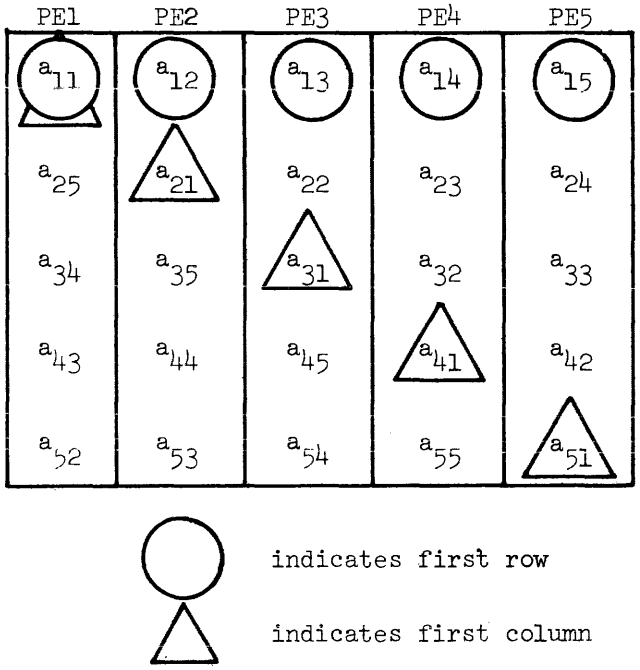


Figure 4 – Skewed Storage Technique for Matrices

of a column, neighboring PE's access elements differing in address by 1.

The illustrated matrix fits nicely into the memory allotted. For matrices where this is not true packing schemes exist to minimize wasted storage space in relation to the operation being performed.

The case of skewed storage just described is an important but simple example of the unique considerations made necessary by the parallel organization of this computer. Manual calculation (like man himself) is essentially sequential. Our electronic calculating aids (computers) have hewn strictly to this "natural" organization of calculations. It is, however, by no means "natural" for a technological innovation to so restrict itself. In fact, man is a slow, highly error prone computing device. The parallel approach to computing, it must be said however, does require that some original thinking be done about numerical analysis and data management in order to secure efficient use. In an environment which has represented the absence of the need to think as the highest virtue this is a decided disadvantage.

Validity of the single processor approach to achieving large scale computing capabilities

by DR. GENE M. AMDAHL
International Business Machines Corporation
Sunnyvale, California

INTRODUCTION

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. Variously the proper direction has been pointed out as general purpose computers with a generalized interconnection of memories, or as specialized computers with geometrically related memory interconnections and controlled by one or more instruction streams.

Demonstration is made of the continued validity of the single processor approach and of the weaknesses of the multiple processor approach in terms of application to real problems and their attendant irregularities.

The arguments presented are based on statistical characteristics of computation on computers over the last decade and upon the operational requirements within problems of physical interest. An additional reference will be one of the most thorough analyses of relative computer capabilities currently published – "Changes in Computer Performance," *Datamation*, September 1966, Professor Kenneth E. Knight, Stanford School of Business Administration.

The first characteristic of interest is the fraction of the computational load which is associated with data management housekeeping. This fraction has been very nearly constant for about ten years, and accounts for 40% of the executed instructions in production runs. In an entirely dedicated special purpose environment this might be reduced by a factor of two, but it is highly improbably that it could be reduced by a factor of three. The nature of this overhead appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor. The non-housekeeping part of the problem could exploit at most a processor of performance three to four times the performance of the housekeeping processor. A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

Data management housekeeping is not the only problem to plague oversimplified approaches to high speed computation. The physical problems which are of practical interest tend to have rather significant complications. Examples of these complications are as follows: boundaries are likely to be irregular; interiors are likely to be inhomogeneous; computations required may be dependent on the states of the variables at each point; propagation rates of different physical effects may be quite different; the rate of convergence, or convergence at all, may be strongly dependent on sweeping through the array along different axes on succeeding passes; etc. The effect of each of these complications is very severe on any computer organization based on geometrically related processors in a paralleled processing system. Even the existence of regular rectangular boundaries has the interesting property that for spatial dimension of N there are 3^N different point geometries to be dealt with in a nearest neighbor computation. If the second nearest neighbor were also involved, there would be 5^N different point geometries to contend with. An irregular boundary compounds this problem as does an inhomogeneous interior. Computations which are dependent on the states of variables would require the processing at each point to consume approximately the same computational time as the sum of computations of all physical effects within a

large region. Differences or changes in propagation rates may affect the mesh point relationships.

Ideally the computation of the action of the neighboring points upon the point under consideration involves their values at a previous time proportional to the mesh spacing and inversely proportional to the propagation rate. Since the time step is normally kept constant, a faster propagation rate for some effects would imply interactions with more distant points. Finally the fairly common practice of sweeping through the mesh along different axes on succeeding passes poses problems of data management which affects all processors, however it affects geometrically related processors more severely by requiring transposing all points in storage in addition to the revised input-output scheduling. A realistic assessment of the effect of these irregularities on the actual performance of a parallel processing device, compared to its performance on a simplified and regularized abstraction of the problem, yields a degradation in the vicinity of one-half to one order of magnitude.

To sum up the effects of data management housekeeping and of problem irregularities, the author has compared three different machine organizations involving approximately equal amounts of hardware. Machine A has thirty two arithmetic execution units controlled by a single instruction stream. Machine B has pipelined arithmetic execution units with up to three overlapped operations on vectors of eight elements. Machine C has the same pipelined execution units, but initiation of individual operations at the same rate as Machine B permitted vector element operations. The performance of these three machines are plotted in Figure 1 as a function of the fraction of the number of instructions which permit parallelism. The probable region of operation is centered around a point corresponding to 25% data management overhead and 10% of the problem operations forced to be sequential.

The historic performance versus cost of computers has been explored very thoroughly by Professor Knight. The carefully analyzed data he presents reflects not just execution times for arithmetic operations and cost of minimum of recommended configurations. He includes memory capacity effects, input-output overlap experienced, and special functional capabilities. The best statistical fit obtained corresponds to a performance proportional to the square of the cost at any technological level. This result very effectively supports the often invoked "Grosch's Law." Utilizing this analysis, one can argue that if twice the amount of hardware were exploited in a single system, one could expect to obtain four times the performance. The only difficulty is involved in knowing how to exploit this additional hardware. At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome. If it were easy they would not have been left as bottlenecks. It is true by historical example that the successive obstacles have been hurdled, so it is appropriate to quote the Rev. Adam Clayton Powell—"Keep the faith, baby!" If alternatively one decided to improve the performance by putting two processors side by side with shared memory, one would find approximately 2.2 times as much hardware. The additional two tenths in hardware accomplishes the crossbar switching for the sharing. The resulting performance achieved would be about 1.8. The latter figure is derived from the assumption of each processor utilizing half of the memories about half of the time. The resulting memory conflicts in the shared system would extend the execution of one of two operations by one quarter of the execution time. The net result is a price performance degradation to 0.8 rather than an improvement to 2.0 for the single larger processor.

Comparative analysis with associative processors is far less easy and obvious. Under certain conditions of regular formats there is a fairly direct approach. Consider an associative processor designed for pattern recognition, in which decisions within individual elements are forwarded to some set of other elements. In the associative processor design the receiving elements would have a set of source addresses which recognize by associative techniques whether or not it was to receive the decision of the currently declaring element. To make a corresponding special purpose non-associative processor one would consider a receiving element and its source addresses as an instruction, with binary decisions maintained in registers. Considering the use of thin film memory, an associative cycle would be longer than a non-destructive read cycle. In such a tech-

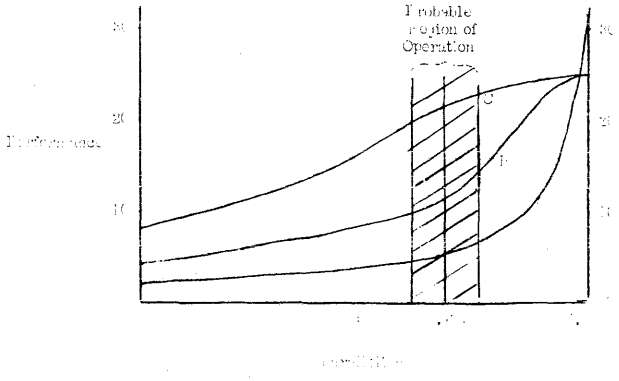


Figure 1

nology the special purpose non-associative processor can be expected to take about one-fourth as many memory cycles as the associative version and only about one-sixth of the time. These figures were computed on the full recognition task, with somewhat differing ratios in each phase. No blanket claim is intended here, but rather that each requirement should be investigated from both approaches.

Analog & hybrid computers in education: a panel session

by Dr. LADIS KOVACH, Chairman
Mathematics & Physics Department
Pepperdine College
Los Angeles, California

Since the analog computer is ideally suited to solving differential equations, its use in teaching that subject is well established. With the advent of inexpensive, reliable, and accurate function generators and multipliers the analog computer can also be effectively used to teach other topics in mathematics.

A number of examples are given ranging from algebra to complex variables. It is shown how the analog computer helps a student understand various concepts by giving a dynamic representation. The pedagogy is improved when the student himself changes parameters and investigates the different cases that arise. In this way he may also make some discoveries which will further strengthen his understanding.

One of the unusual examples given illustrates how the analog computer can be used to teach numerical analysis. Another shows how the teaching of linear programming can be accomplished. The great value of the analog computer in teaching various branches of mathematics to liberal arts students is stressed. Thus some of the mystery surrounding mathematics can be dispelled for these students.

by DR. L. E. BURKHART
Department of Chemical Engineering
Iowa State University
Ames, Iowa

In simulation, the analog flowsheet preserves the correlation between the computer program and the physical system which it represents. Identification of each part of the process which is simulated is simple and direct. The analog computer, being essentially a parallel instrument, provides rapid solutions and permits instantaneous parameter changes. Such a system has great educational value by providing a "live" simulation which can be studied and varied during solution of the problem.

Hybrid computers should be emphasized in those areas where they have definite superiority over analog and digital computers--for example, in optimizing and search procedures. New developments in hybrid software will provide better access to hybrid machines and their advantages in certain areas will be much easier to exploit.

At whatever educational level analog computation is introduced, students must feel that use of the computer has actually contributed to an easier or a better solution to the problem at hand. Thus, motivation of students at the college junior level and above is excellent because they have had enough science and mathematics that the computer can be used to solve meaningful problems in their particular field. Below this level, student motivation must be more carefully considered, and the character of the problems involved becomes more important.

by DR. ROBERT KOHR
Department of Mechanical Engineering
Purdue University
Lafayette, Indiana

Analog and, more recently, hybrid computers are playing an increasingly important role in the education of mechanical engineers. This discussion contains a description of the use of analog and hybrid computers as tools in the education of mechanical engineers at Purdue, particularly in the areas of Dynamics, Instrumentation, Heat Transfer, Fluid Mechanics, and Automatic Control. A detailed description of an undergraduate course in Systems Analysis and Control Employing Analog Computers for the simulation of the dynamic response of feedback control systems will be given. Typical experiments, lecture demonstrations, and student response

to them will be discussed. The use of high speed, interative analog and hybrid computation in graduate level courses in Mechanical Engineering will also be described.

by DR. MYRON L. CORRIN
Department of Chemistry
University of Arizona
Tucson, Arizona

The use of analog computation in chemical research and the teaching of chemistry at the undergraduate and graduate levels will be discussed. In research the user must understand the capabilities and limitations of computer techniques and must be able to program his problems. Research applications in chemistry have been limited almost completely to chemical kinetics. Some such applications will be considered.

In the teaching of chemistry it is not necessary that the student understand computer operation; the instrument may be used as a "black-box" which will solve certain problems. It is necessary that the student understand graphical representation. We have employed analog computer demonstrations in courses in freshman chemistry, freshman honors chemistry and in both graduate and undergraduate physical chemistry. It has been noted that about 10 to 15% of students who participate in such demonstrations request additional training in computer use.

Demonstrations in the areas of kinetics, equilibrium, thermodynamics, quantum mechanics and surface chemistry will be discussed. Attention will also be given to the mode of presentation and to means used to induce the student to take an active role. The question of arousing teacher interest has not yet been solved.

by DR. AVRUM SOUDACK
Department of Electrical Engineering
University of British Columbia
Vancouver, British Columbia

I would like to address myself to defining the objectives of analog computer use in undergraduate education. Much has been said about their use in engineering, vs. non-engineering. There seems to be much controversy about the level at which the computer should be introduced and the level of sophistication of the machine. The controversy goes round and round - without really pinpointing objectives. Let me attempt to do this --

Objective 1. The use of the computer as a teaching aid.

Objective 2. The use of the computer as a problem-solving tool.

These are two *widely differing* objectives, each requiring a rather different viewpoint.

In a nutshell, I'd elaborate on the following:

Objective 1.

(a) The black-box approach - right down into highschool.
(b) The machine's potential use for teaching math, physics, perhaps economics, chemistry, any subject involving dynamic changes.
(c) Computer-design for such use.
(d) Degree of elaboration in engineering courses. i.e. - our experience at U.B.C.

Objective 2.

(a) Junior and senior level. Graduate level.
(b) Degree of machine sophistication.
(c) Man-machine rapport at low cost.
(d) Optimization and design. Development of intuition.
(e) Own experience at U.B.C.

I would close with a plea to manufacturers to have these objectives in mind when developing educational machines.

by MR. A. I. KATZ
174 Philip Road
New Brunswick, New Jersey

During the past fifteen years the engineering school teacher has been confronted by the challenge to prepare the engineering school graduate to meet the technical requirements of the Mid-Twentieth Century. Those teachers who responded effectively to this demand were the most creative individuals and, as a result, established their reputation in the engineering community by virtue of the contribution they made in formulating effective engineering curricula.

The results of these efforts were a transition from "Handbook Engineering" to the use of a more scientific approach in engineering. Specifically, this has meant:

1. Greater emphasis on courses in mathematics
2. An increased role of analysis of physical systems
3. A reduction in utilization of empirical formulas
4. A reduction in detailed design effort engineering courses.

These factors have provided a strong foundation for the engineering graduate allowing him to work in many disciplines. Unfortunately, the swing of the pendulum in engineering education has tended to result in an emphasis on the preparation for graduate school, rather than in developing a student who

would enter the engineering field upon receipt of his Bachelor's Degree. In many cases, engineering educators have done little or no engineering design, nor have they had industrial experience and this has often resulted in the failure to relate the analytical techniques to the solution of engineering problems.

As we enter the latter half of the Twentieth Century, we find that engineering systems are growing more extensive and complex. There is a growing need to integrate mechanical, electrical, chemical, etc. systems into an economic and practical package. In view of this trend, the engineering educator must provide the engineering graduate with a better realization and understanding of the "systems approach" to engineering.

The most effective means for developing a "systems approach" is the technique of simulation. (The role of simulation in the Aerospace field is an excellent example of an integrated approach to solving complex problems.)

The key element in engineering simulation is an analog or hybrid computer which has a one-to-one relation to the physical system being modeled. These computers inherently provide the OARS in approach which is the keystone of simulation:

OBSERVE
ABSORB
REACT
SEARCH

The success of systems analysis in large and complex organizations hinges upon the speed of communication between man and machine in a simulation, since simulation is a learning and experimenting process. The ease of making structural changes in a simulation, relating simulation elements to physical system elements and the convenience of interpreting the output of a simulator are essential for effective systems analysis.

The engineering educator needs to utilize analog and hybrid computers to bring to the engineering student the concepts of systems engineering. In addition, he must provide an understanding of these tools, so that the engineering graduate may utilize these computers to their best advantage in solving modern engineering problems after graduation.

DTPL push down list memory*

by R. J. SPAIN, M. J. MARINO, and H. I. JAUVTIS
Laboratory for Electronics, Inc.
Boston, Massachusetts

INTRODUCTION

Push down lists (PUDL) or First In Last Out memories have been described for use in multiprocessors.[1] However, in most of these applications, the system's fast access memory is divided in stacks, each program being run in a given stack on a first stack available basis. The push down list is then used to store the program instructions. Very seldom has such a system been described using more than two to six fast access lists.[2] These applications might not justify the use of a special device for the PUDL, making use, instead, of a random access memory section addressed by a pointer address register or counter to keep track of the upper word in the PUDL. There the PUDL storage is only a small fraction of the total fast access storage, and software implementation is more economical than special hardware would be. However, in more sophisticated data processing applications such that language translation, pattern recognition, high speed data acquisition and retrieval, etc., an efficient system organization would result from the use of a central processor working on a multiplicity of programs on a fast interrupt and logic priority basis. The next active program location would depend upon the output from the presently processed instruction. Then the program storage is an important part or even the largest part of the total fast access storage of the system. In this situation, a large number of lists could be required with an access time of the order of one microsecond. If the program instruction lists are stored in a conventional random access memory one pointer register per list would be required and result in an expensive hardware. The pointer could be stored in memory, however, this would result in much increased access time and additional software. A more economical and more efficient implementation of such a multiprogram organization may result from the use a specialized hardware such as a DTPL[3,4] magnetic thin film Push Down List Memory. The system described here has as an objective the development of a DTPL push down list memory consisting of a set of 128 lists. Each list is 100 words deep with 38 bits per word and thus is implemented by 38 shift registers of 100 bits each. An additional shift register of the same length stores a bottom-of-list (BOL) flag that can be sensed at both ends of the register to deliver an empty or filled list warning signal. Only one magnetic thin film of approximately one inch square area is required for the 38 shift registers constituting one list. The system makes use of a stack of 100 magnetic thin films. Every list is controlled by a conductor that can be selected in a manner similar to that of a word line of a conventional random access memory. The DTPL[3,4] Push Down List Memory makes use of the controlled propagation of elongated domains of reversed magnetization contained within low coercive force channels imbedded in a magnetic thin film of higher coercive force. This technique of implementation is particularly suited to the construction of bi-directional shift registers with non-volatile storage and are operable with shifting rates above one megacycle.

DTPL PUDL implementation

Comprehension of the DTPL implementation of a PUDL memory requires an understanding of the basic functioning of the DTPL shift register. The domain tip propagation mechanism is simple enough to allow understanding from the following abridged description. Figure 1 depicts the low coercive force channel pattern impressed into a high coercive force magnetic thin film background. The shift register channel pattern is a simple zig-zag with at 10° to 30° slant angle with the easy axis. Channel widths are typically 0.002" and the lengths of the zig-zag

*This program is sponsored by the U. S. Army Electronics Command, Fort Monmouth, New Jersey, under a special task of the research in Ferromagnetic program supported by Air Force under Contract AF 19(628)-4197, as well as Laboratory For Electronics, Inc., Boston, Massachusetts.

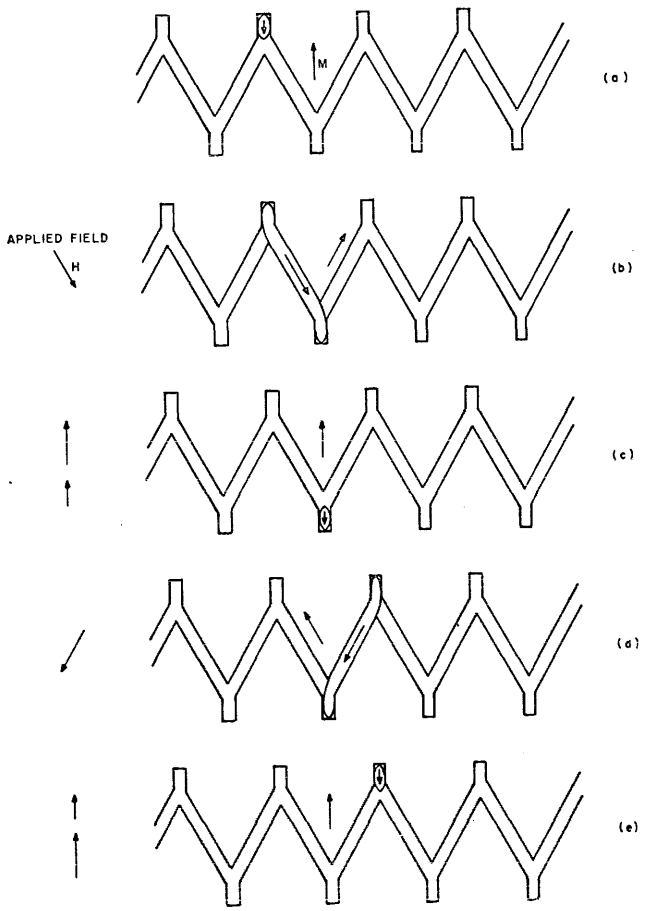


Figure 1 – The operation of a domain tip propagation shift register – The direction (↘↙,↓) and relative strength (↑↑ of the applied fields are shown – The easy axis is vertical

channel segments, 0.015". Channel tabs at each corner are used to store the data word in the form of a small domain of reversed magnetization in the upper tabs for each bit. The magnetization vector in the rest of the channel is parallel with the uniform magnetization of the background film, along the easy axis. The field applied to produce growth of the information bearing domains is a combination of easy and hard axis field which define the axis along which domain tip propagation is favored. By alternating the sense of the hard axis component of the field applied during the growth pulses, unidirectional propagation of information is achieved. The process is illustrated in Figure 1. A small domain of reversed magnetization is initially situated at an intersection of adjacent channel segments of a long zig-zag low coercive force channel as in part (a) of the figure. A field is applied as in (b) such as to cause propagation of the domain tip diagonally through one channel segment. A restoring field as in (c) is applied causing erasure of the domain everywhere within the channel except at the channel intersection diagonally opposite the initial bit location. A growth pulse follows this time with the sense of the hard axis field reversed. The result is shown in (d). A restoring field is applied, as in (e), erasure again being inhibited at the new intersection of the channel segments. This completes the shifting cycle and it may be seen that the original reversed domain has been displaced one bit location to the right. Propagation of this sort occurs only for domains of reversed magnetization initially present at bit locations. Spontaneous nucleation of extraneous domains does not occur; nor is there any loss of existing domains. Means for preventing the complete erasure of propagated domains during the restoring pulsing [see Parts (c) and (d) of the figure] consists of using inhibit conductor lines which run beneath the bit locations. The control of the shifting operation is quite simple. An inhibit line placed in close proximity and across the upper tabs of the zig-zag low coercive force channels returns across the lower tabs. When pulsed with one polarity of externally applied erase field, the inhibit line preserves the information domains under the upper tabs and enhances erasure under the lower tabs. Conversely, when the inhibit line is pulsed with the opposite polarity of current, the information domains are preserved under the lower tabs, and enhanced erasure occurs under the upper tabs. Thus, by the use of alternate polarities of current in the inhibit line, shifting of the information occurs between the upper and lower tabs. Shift to the right or to the left is controlled by either the polarity sequence of the inhibit current or by the polarity sequence of the hard axis component of the applied field during the domain growth time.

A sequence of shifting operations consists of alternating the propagate and erase fields. Introduction of an additional conductor per register permits controlling the advancing of one shift register among a set of registers. This additional conductor is placed in close proximity with the segments of the inhibit conductor crossing the upper tabs of the zig-zag patterns. When this additional control line is pulsed simultaneously with, but in opposition to the inhibit line, cancellation of the inhibit conductor action results. The shift register advance selection is described as follows: Figure 2 illustrates the conductor routing in relation with the zig-zag low coercive force channel pattern. Line K, called the keep line, crosses only the upper tabs of all registers in the same direction but avoids the first tab that is used for read out only. Line K is pulsed during every erase time so that the information stored in the lists is normally kept at the same location in all lists. Line T called transfer line crosses alternately the upper tabs and the lower

tabs in opposite directions for every register of a given list. Line T is pulsed only in the selected list during the first tip propagation time of a clock cycle. Pulsing line T with the proper current polarity allows cancelling the keep action in the upper tab and transferring information in the lower tabs. At the next propagation time within the same clock cycle, the tip propagation is upwards and the reversed magnetization domains are subsequently kept in the upper tabs, thus terminating shift to the left or to the right, depending upon the polarity of the hard axis component of the applied field.

Mode pulse sequences are shown on the timing diagram depicted in Figure 3, where a few more particularities should be noted:

1. Write data is operated through domain nucleation to the first lower tab of a given register by coincidence of the transfer signal and the write data bit signal during the first erase time of a write and push down cycle. This coincident writing mode allows list selection for the write operation.
2. Erase list is operated by keeping cancellation intervening just after the reversed magnetization domains have been erased in the lower legs during the beginning of an extended erase time. Memory clear would be operated by a unique external field erase pulse and would require 300 nanoseconds. Memory clear can also be achieved during an operating cycle by inhibiting the keep function.

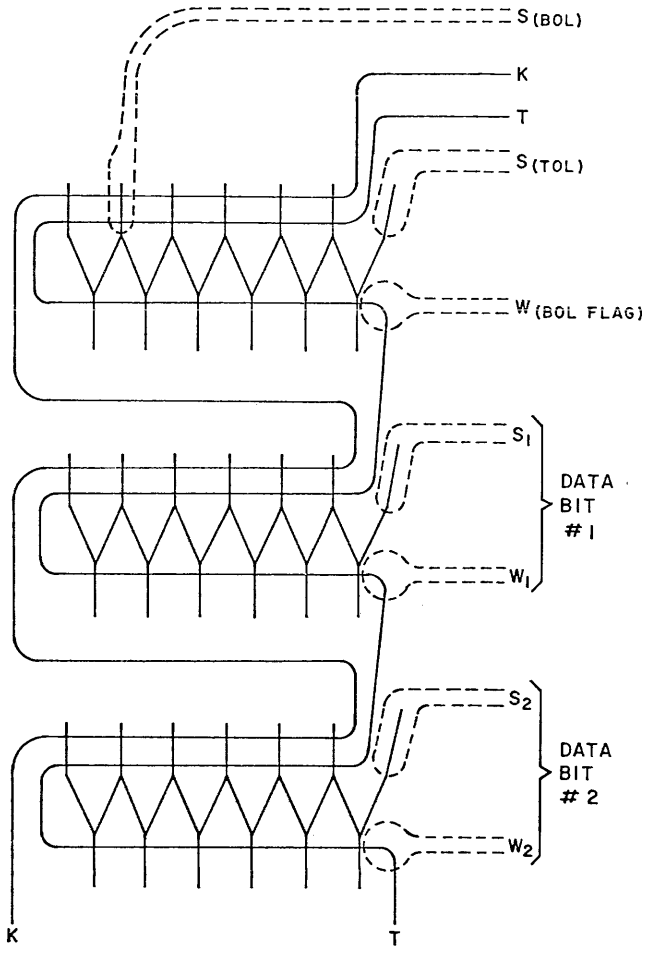


Figure 2 – Conductor pattern in relation with the magnetic thin film low coercive force channels

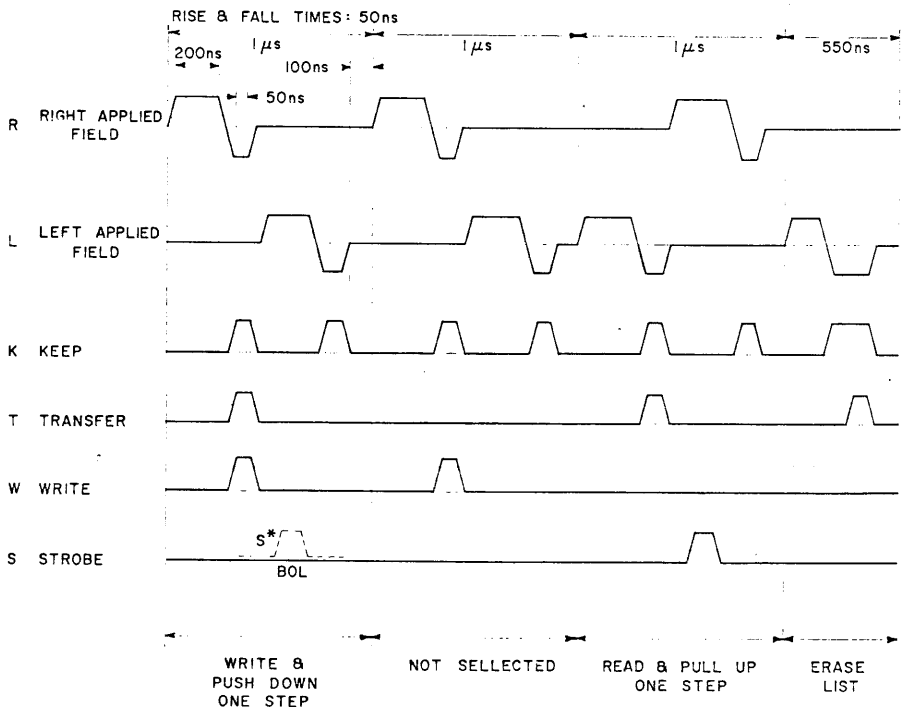


Figure 3 – Timing diagram

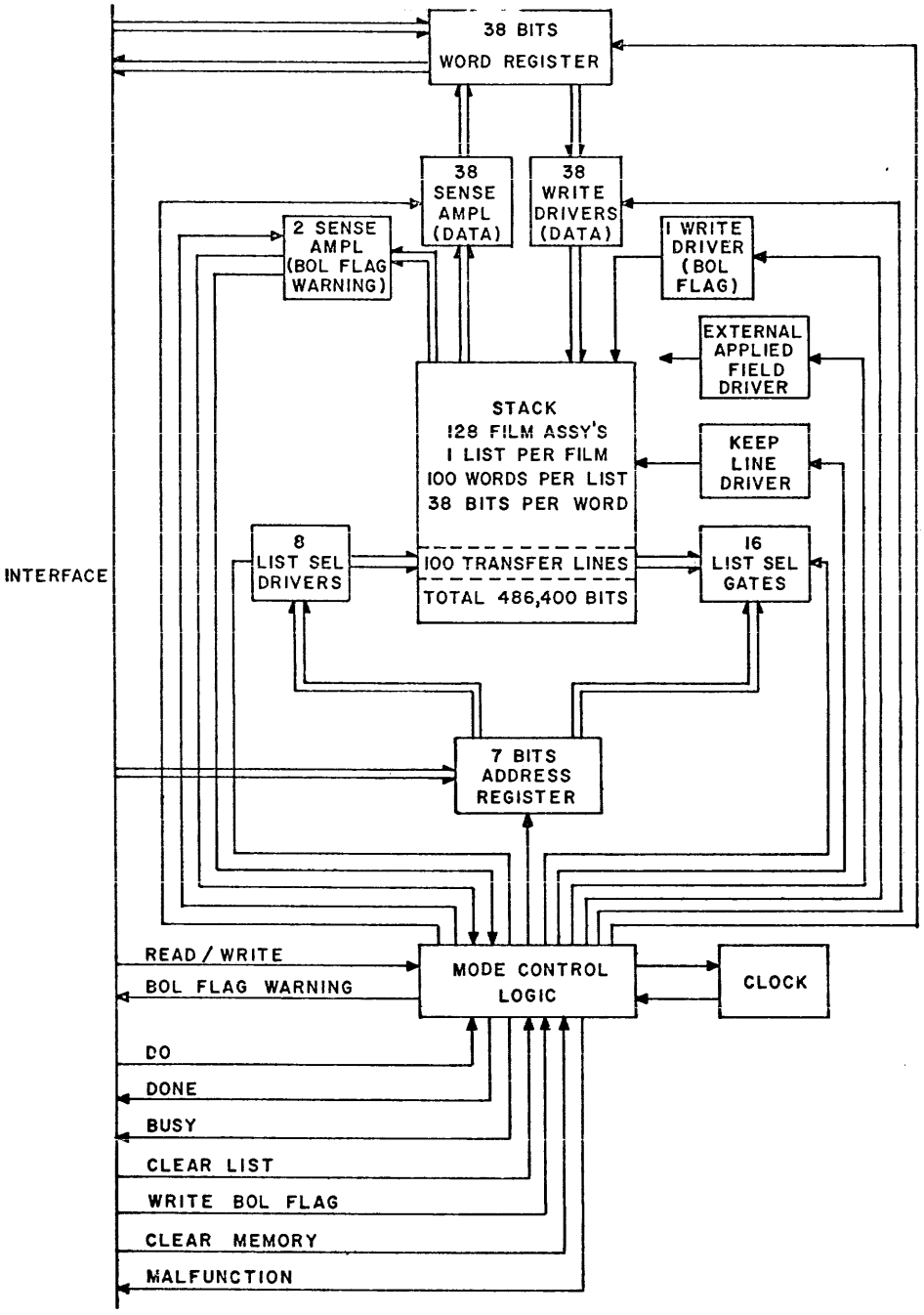


Figure 4 – DTPL PUDL block diagram for 128 lists

3. Strobe signal S is used during the second propagation time of the Read and pull up cycle when the domain tip passes under the pick up loop conductor S shown on Figure 2. This read out is destructive since it is not desired to rewrite a just read-out word.

4. The BOL strobe S* occurs at the same clock slot time as for the data strobe, however, during a write and push down cycle, since it corresponds to a warning signal located at the bottom of the list. This BOL signal is a flag bit located on an additional register per list and can be written at any time or erased simultaneously with the content of its associated list. The BOL pick up loop is located a few bits before the end of the register.

5. Any shifting cycle lasts one microsecond including a blank space of 100 nanoseconds for change of list address and mode controls.

System organization

A block diagram for the DTPL push down list is shown in Figure 4. The stack consists of 128 lists

with a total capacity of 486,400 data bits. The cycle time is 1 μsec for either write and push down one step or read and pull up one step. The stack is accessed in parallel by words of 38 bits and thus requires 38 sense amplifiers and write current drivers. One list is accessed at a time through a linear selection scheme simplified by the fact that the current pulses required to select a list through its transfer lines are unipolar. Only one steering diode per list is required and the line is selected by coincidence of conduction of a list selection driver on one side of the line and of a list selection gate at the other end of the line. The optimum number of drivers and gates is $\sqrt{N}$ for a N lists memory. For 128 lists, 8 drivers and 16 gates will be employed, resulting in a very economical selection electronics. Two additional drivers are required in the system, a general right-left external applied field driver and a keep line driver. Address register and Word register are similar to those found in conventional random access memories and they include the ordinary in-out gates. The clock consists of a string of monostable multivibrators generating a pulse sequence utilized in the various current drivers and strobing circuits. The mode control logic network is accepting signals from the interface: (1) a dc logic level for READ-WRITE, (2) CLEAR LIST pulse to order erasure of the data and of the BOL flag contained in the list selected by the address, (3) WRITE BOL FLAG pulse to write a flag bit in the top of the BOL shift register of the list selected by the address, during a write and push down cycle, (4) CLEAR MEMORY pulse to initiate a half clock cycle without pulsing of either the Keep or Transfer lines resulting in erasure of all data contained in the memory. The Mode Control Logic network is also sending memory status information to the interface such as: (1) DONE pulse, occurring just after utilization of the orders received from the interface, (2) BUSY level remaining high until execution of the orders, (3) BOL FLAG WARNING pulse when the BOL flag reaches either the BOL warning locations, near the bottom of the selected list during a write cycle, or the top of the list during a read cycle, (4) MALFUNCTION logic level normally low and becoming high when either a parity error or some other malfunction is detected. The latter is an optional feature not essential to the PUDL mechanism and dependent upon system application requirements. The Mode Control Logic netwrok is small as it consists only of 4 set-reset flip flops and 15 two input gates.

Feasibility model

A reduced size system feasibility model is presently being built. This model consists of 8 lists with 25 words per list. There are 13 bits per word plus the BOL bit. But for its reduced size, the feasibility model is similar to the 128 list DTPL PUDL memory described above, particularly, the cycle time is also one microsecond, clock and mode control logic are identical to what should be used in the final system. All circuits have been breadboarded and do not offer any particular novelty; they are standard high speed linear selection memory type circuits and do not need to be described here. However, initial design and breadboarding of the external applied field drivers required considerable work because of the large transient voltages encountered during the current transitions in the large inductances employed at first. A novel design makes use of flat etched coil pairs placed on each side of each film assembly and driven as transmission lines. Ten such coils are connected and driven in series, corresponding to a delay smaller than fifty nanoseconds. Thus for 100 lists, the external applied field driver final stages will be driven in parallel for both the R and the L applied fields. Figure 5a illustrates the flat coils and the DTPL PUDL access conductor half assembly. Two double sided copper clad thin epoxy boards are etched according to 4 patterns; starting from the bottom layer, one can observe the hard axis field coil, the easy axis field coil with orthogonal windings, the keep line, the sense and write conductors are located on the top layer.

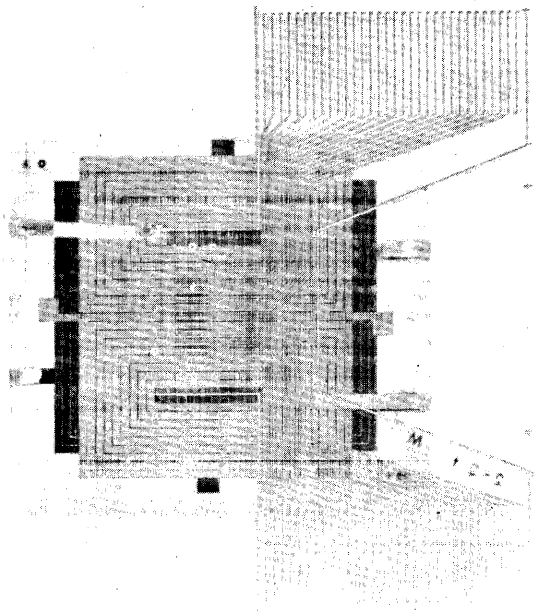


Figure 5 – PUDL conductor pattern

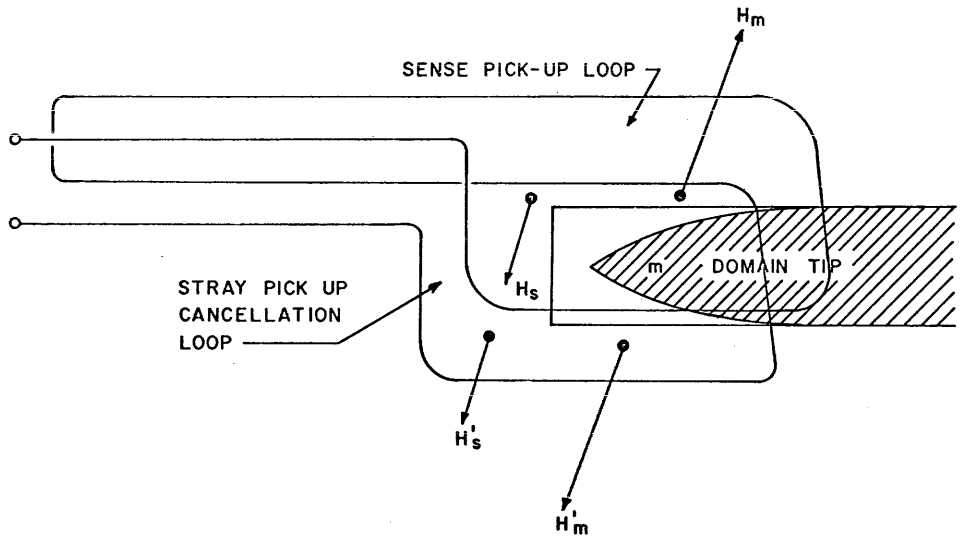


Figure 6 – Relative positions of the read out conductors and the read out channel

The magnetic thin film substrate, presently a two to three mil thick glass slide, is placed in registration in the center of the coil assembly. The applied field created by the flat coils is uniform within the magnetic film area. Then another conductor assembly is prepared similarly to the one just described. All conductor patterns are mirror images of the first assembly but for substitution of the Keep line by the Transfer line. Where there is imperfect alignment of the thin film element with respect to the axis of symmetry of the drive coils. The pairing of sense line conductors allows cancellation of the stray pick up and a doubled output signal in the sense line as is illustrated in Figure 6. The magnetic charge m found near the end of the domain tip creates opposed magnetic fields Hm and H'm in the pick-up loop and its mirror image. Therefore, when the domain tip travels across the pick up loop the two induced voltages, $\frac{d(\phi m)}{dt}$ and $\frac{d(\phi' m)}{dt}$ add at the input to the sense amplifier. However, the residual vertical component of the external field is practically uniform both loops and cancellation of the stray pick up is obtained. Thus, the inductance of these coils is minimized and they behave as strip transmission lines. A complete film assembly would be about 20 mils thick and such assemblies can be stacked at less than 100 mils centers resulting in a compact memory stack.

Complete system measurements are not available, however, very little operation margin variations have been observed from film to film. Figure 7 illustrates a plot of the working area for an experimental DTPL PUDL shift register operated with 220 nanoseconds propagation times and 50 nanoseconds erase times at 90% of the pulse maximum amplitude. The applied field amplitudes along the easy axis and hard axis are indicated in Oersteds and were evaluated from actual driving currents. The dotted line corresponds to device operation limited by spontaneous nucleation of domains under the keep conductor at a given hard axis field level. The tolerances on all applied currents is±16% and believed to be sufficient

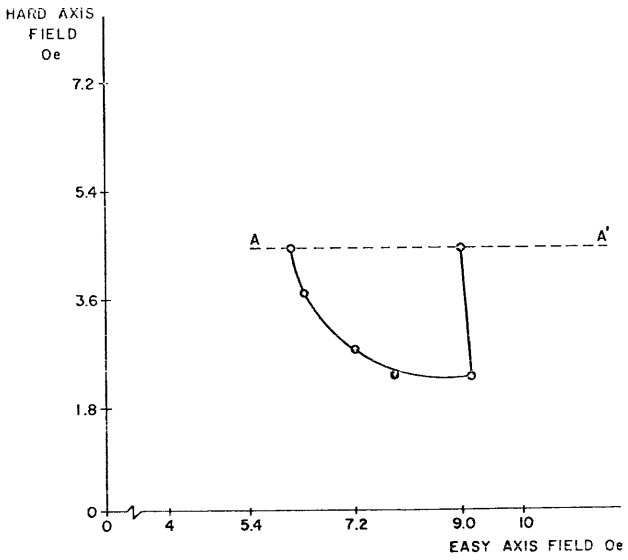
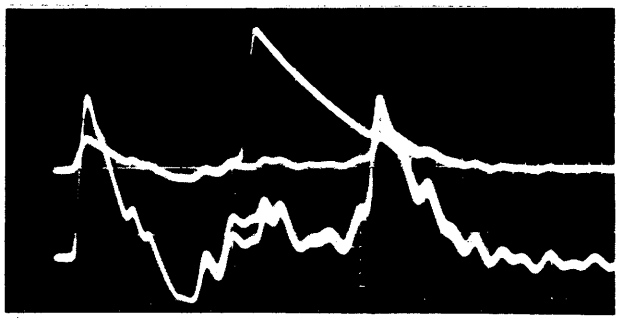


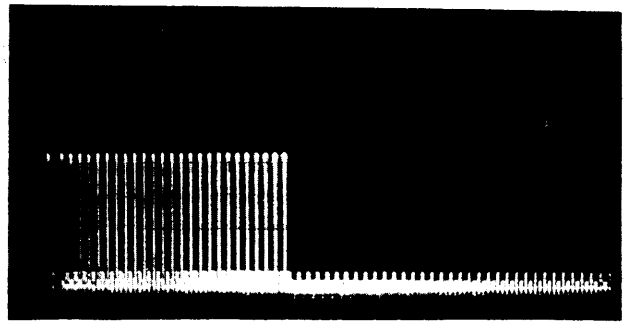
Figure 7 – Area of reliable functioning for a IMC DTPL shift register

for reliable operation of the PUDL memory when using a single ±5% regulation of the currents in the selection circuitry.

Figures 8 and 9 show the operation of a 26 bit shift register channel. Figure 8a depicts the strobed (top trace) and unstrobed (bottom trace) signals in a condition of pushing down and pulling up "ones" and "zeros." The horizontal scale is 200 ns/cm. The vertical scale is 1.0 v/cm in the case of the strobed signal and 200 mv/cm in the case of the unstrobed signal. Figure 3b shows the result of pushing down and pulling up more than 26 "ones." Since the capacity of these registers is 26 bits, only 26 "ones" are seen to return. Thereafter only "zeros" are read out. The gap seen following the return of the first "one" is a result of the programming sequence utilized. The cycle time is approximately 2 μsec.



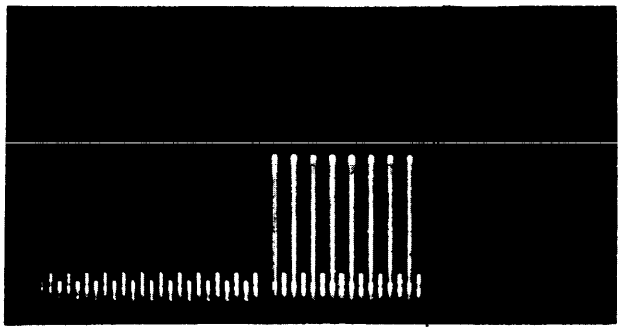
(a)



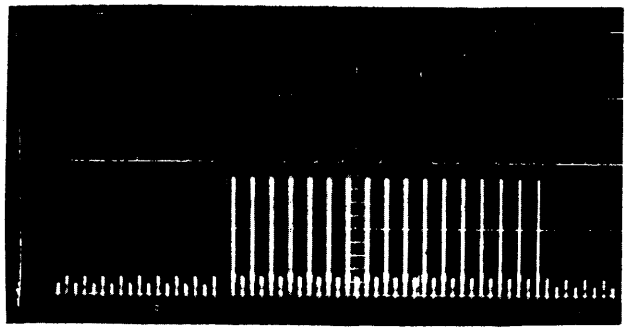
(b)

– Signal outputs from DTPL shift register

Figure 9a shows the result of pushing down 12 "zeros" and 8 "ones" and then pulling them up. In Figure 9b, 9 "zeros" were introduced into the registers and then followed by more than 17 "ones." In pulling the information back up the register, only 17 "ones" are seen to return after the "zeros," again indicative of correct operation of the 26 bit register.



(a)



(b)

Figure 9 – Signal outputs from DTPL shift register

CONCLUSION

The conductor assembly described here is not the only possible implementation of the DTPL list selection. Actually, better line electrical characteristics could be obtained from the use of single inhibit line with mirror image return on the other side of the magnetic thin film instead of using separate keep and transfer lines. However, this conductor pattern would result in an increased selection circuitry since both the external applied fields through the flat coils and the inhibit lines would have to be selected with bipolar currents. Improvements of the readout method are also under study as the output from the sense line is presently in the 50 microvolt-150 nanosecond range and requires two stages of integrated circuit sense amplifier amplification where

the compensation of the input offset voltage becomes difficult to obtain over a large temperature range. More system application study is also required as it seems that the real economy of the DTPL PUDL implementation lies in the use of a large number of lists of up to 200 words per list where only one list is accessed from the top at a time on a random access mode with 1 μsec access time and cycle time. It is felt that such characteristics are quite desirable but have not been considered in reported programs or investigations probably because of lack of hardware implementation.

ACKNOWLEDGMENT

The authors are grateful to David Haratz, Lorenz M. Sarlo and David R. Hadden of Fort Monmouth for their contributions to the DTPL PUDL original concept.

The authors also want to acknowledge the help of the staff of the Applied Research Department of the Electronics Division of Laboratory for Electronics, Inc. for the work reported here. Particularly valuable have been the contributions made by C. P. Battarel.

REFERENCES

1 A narrative description of the Burroughs B5500 Disk File Master Control Program-Burroughs Corporation.

2 H J GRAY C KAPPS ET AL
Interactions of computer language and machine design
DDC Defense Supply Agency Clearinghouse AD 617 616

3 R J SPAIN
Domain tip propagation logic
IEEE Trans on Magnetics vol MAG 2 no 3 September 1966

4 R J SPAIN H I JAUVTIS
Controlled domain tip propagation
J Appl Phys vol 37 no 7 June 1966

An integrated MOS transistor associative memory system with 100 ns cycle time

by RYO IGARASHI and TORU YAITA
Nippon Electric Company Ltd.
Tokyo, Japan

INTRODUCTION

Since the announcement of the development of a technique for using MOS transistor integrated circuits as associative memory cells,[1] 128 words of 48 bits per word associative memory has been experimented and engineered.

This paper deals with characteristics of a fully integrated associative memory chip utilizing MOS transistor technology and a method of a multimatch resolving in the associative memory system.

The associative memory chip contains 4 associative memory cells encapsulated in a 14-lead flat package.

This associative memory cell is designed so as to operate at a cycle time of 100 ns with standby power less than 100 μw and power dissipation less than 1 mW (peak) during the interrogate operation.

The advantage of using these associative cells is that they will provide economically practical cells with large scale integration and also they will be useful to realize a large capacity associative memory because of their small power dissipation.

This report is composed of two sections. In the first section, theoretical analysis and experimental results of operating speed of an integrated MOS transistor associative cell are shown. The second section describes the circuit configuration and the method of multimatch resolving of an associative memory matrix.

Associative memory cell

A four bit fully integrated associative memory chip is shown schematically in Figure 1 and a photograph of the associative cell is also shown in Figure 2.

The flip-flop is constructed with the MOS transistors Q_1, Q_2, Q_3, and Q_4. The MOS transistors Q_5 and Q_6 are used for read and write operations, and the MOS transistors Q_7 and Q_8 for interrogate operation. MOS transistors Q_3 and Q_4 are used as load resistors of Q_1 and Q_2, and have characteristics

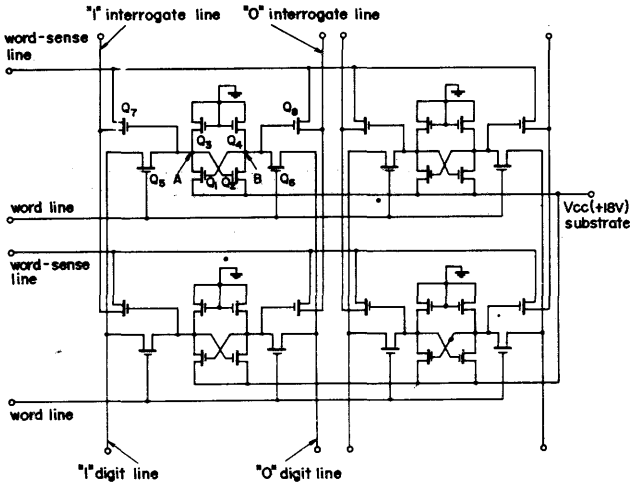


Figure 1 – Schematic circuit of a fully integrated associative memory chip

equivalent to resistance of about 3 MΩ. Q_1 and Q_2 have the highest mutual conductance.

The operation of the associative cell will be described qualitatively at first. The "1" and "0" digit lines are usually held at +5V, the "1" and "0" interrogate lines and the work-sense line at +18V, and the word line at +15V. Then, if Q_1 is on and Q_2 is off (the flip-flop stores a "1"), Q_5 conducts when the voltage level at the word line is shifted from +15V to −5V for reading, resulting in a digit sense current output at the "1" digit line and no output at the "0" digit line. On the other hand, if the flip-flop stores a "0", Q_6 conducts resulting in a digit sense current output at the "0" digit line. Figure 3 shows the timing of waveforms during read and write cycles. During the write cycle the voltage level at the selected word line shifts from +15V to −5V and at the same time the write information is given on the "1" or "0" digit lines as shown in Figure 3.

For an interrogation of "0", the voltage level at the "0" interrogate line is shifted from +18V to +16V. If the flip-flop stores a "0", it is so designed

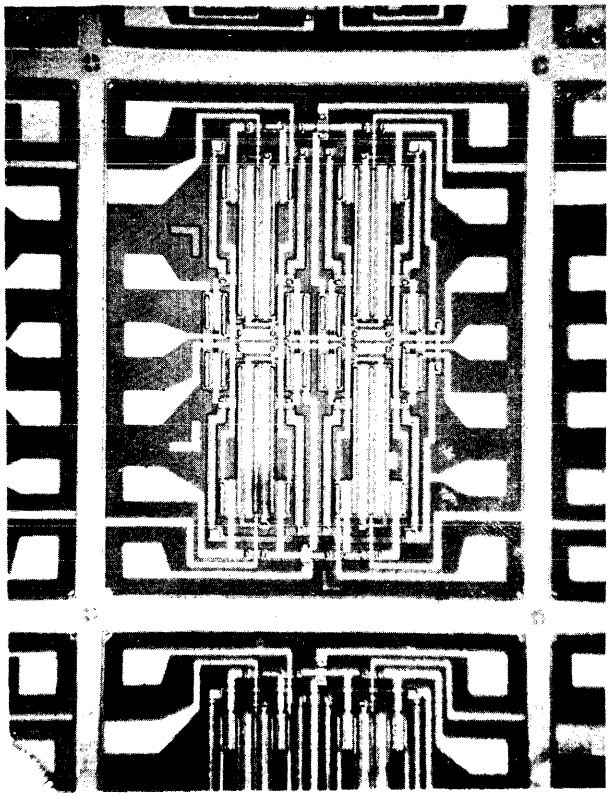


Figure 2 – Photograph of associative cell

that Q_8 is off, therefore no current is supplied to the word-sense line. If the flip-flop stores a "1", a large output current appears on the word-sense line since Q_8 is on.

For an interrogation of "1", large output current is supplied to the word-sense line through Q_7 when the flip-flop stores "0".

In short words, this associative cell generates a signal on the word-sense line only when there is a mismatch between the state of the flip-flop and the interrogate information. And no signal appears when the associative cell is not interrogated (don't care).

The feature of this associative cell is that the power dissipation at Q_7 or Q_8 can be made small when mismatch signal appears in the interrogate operation. Namely, the characteristics of Q_7 and Q_8 are chosen properly so as to give a small voltage between drain and source of Q_7 or Q_8 and to give a large current to the word-sense line.

The current detector composed of an NPN transistor in grounded base configuration is capable of detecting the current given to the word-sense line kept at +18V.

In the worst case, namely in case that a mismatch signal is given by one bit out of 48 bits in one word, the current given to the current detector is to be about half of the current generated at an associative cell.

Another feature of this associative cell is that the mutual conductance of Q_1 and Q_2 is chosen to be three or four times of that of Q_5 and Q_6. This relationship is the necessary condition to prevent from the disturbance of flip-flop in the read mode.

Since Q_3 and Q_4 are actually equivalent to 3 MΩ resistor, as mentioned already, they give almost no effects on the switching characteristics. High-speed write operation is realized by MOS transistors Q_4 and Q_6.

Now provided Q_1 is off and Q_2 is on, let us estimate the switching speed of this associative memory cell in the case of shifting the voltage of "1" digit line from +5V to +Va (+Va > +5V) and the voltage of word line from +15V to some value Vw, resulting in on state of Q_5 and Q_6. In order to make Q_1 on state, first Q_2 is made off by increasing the voltage of node A. When Q_2 gets off state, Q_6 conducts and hence the voltage of node B decreases. When the voltage of node B gets to some value, Q_1 becomes on state. Therefore, after Q_1 reaches sufficient on state, write operation of "1" is finished by restoring the voltage of "1" digit line from +Va to +5V and word line from V_m to +15V. Estimation is made in two steps ; in the first step calculation is done for the time t_1 required to make Q_2 off as a result of increasing the voltage of node A by Q_5, and in the second step the time t_2 required to make Q_1 on as a result of decreasing the voltage of node B by Q_6 is calculated. In order to calculate t_1 and t_2 the characteristics of P-channel MOS transistor are given by the following equations.[2]

$$i_d = k_p \{ 2(V_g + V_T)v_d - v_d^2 \} \quad v_d > V_g + V_T$$

where

i_d	: drain current
k_p	: constant
V_g	: gate voltage for source
V_T	: absolute value of threshold voltage
v_d	: drain voltage for source

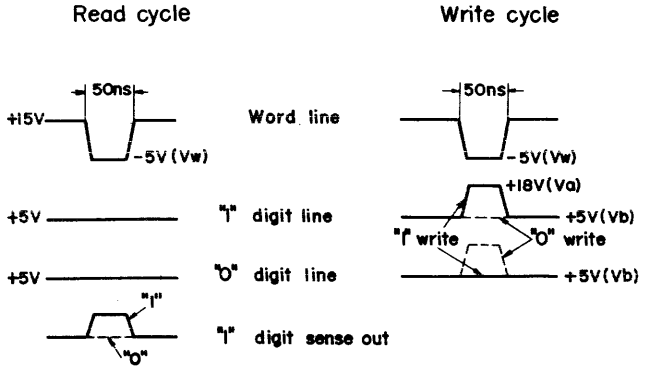


Figure 3 – Timing of waveforms at read and write cycles

The following gives the time t_1 required to shift the voltage of node A from an initial value V_o, which is specified by the threshold voltage of Q_3, to the voltage $V_{cc}-V_T$ to make Q_2 off.

$$t_1 = C_A \int_{V_o}^{V_{cc}-V_T} \frac{dv}{i_{d5}}$$

$$= \frac{C_A}{2K_p(V_w-V_a+V_T)} \ln \left[\frac{(V_{cc}-V_T-V_a)\{2(V_w+V_T)-V_a-V_o\}}{(V_o-V_a)\{2(V_w+V_T)+V_T-V_a-V_{cc}\}} \right]$$

where symbols are as shown in Figure 1 and 3 and i_{d5} is the drain current of MOS transistor Q_5, and C_A is the total capacitance of node A. V_{cc} is shown in Figure 1.

Now the time t_2 required to shift the voltage of node B from V_{cc} to the initial value of node A is given by

$$t_2 = C_B \int_{V_o}^{V_{cc}} \frac{dv}{i_{d6}}$$

$$= \frac{C_B}{2K_p(V_w+V_T-V_b)} \ln \left[\frac{(V_b-V_o)\{2(V_w+V_T)-V_b-V_{cc}\}}{(V_b-V_{cc})\{2(V_w+V_T)-V_b-V_o\}} \right]$$

where i_{d6} is the drain current of MOS transistor Q_6, and C_B is the total capacitance of node B.

Therefore switching time t_w is given by

$$t_w = t_1 + t_2.$$

Figure 4 shows the relation between t_w and C_A ($=C_B$), provided $K_p = 0.05$ mA/V^2, $V_T = 5$V. It is easily found from Figure 3 that t_w becomes 30 ns in case $V_w = -5$V, $C_A = 10$ PF, and $V_o = +7$V and thus the switching time is fast enough to achieve a cycle time of 100 ns.

Figure 5 shows the experimental results of measurement of this associative cell.

Initially, an associative cell is placed in the "1" state by the coincidence of the first word pulse and the "1" write digit pulse, which are shown by traces (a) and (b), respectively. Then, the second word pulse to read the state of the cell follows. The trace (g) corresponding to the second word pulse shows the output for a "1" condition. The cell is next interrogated by the "0" interrogate pulse of trace (d),

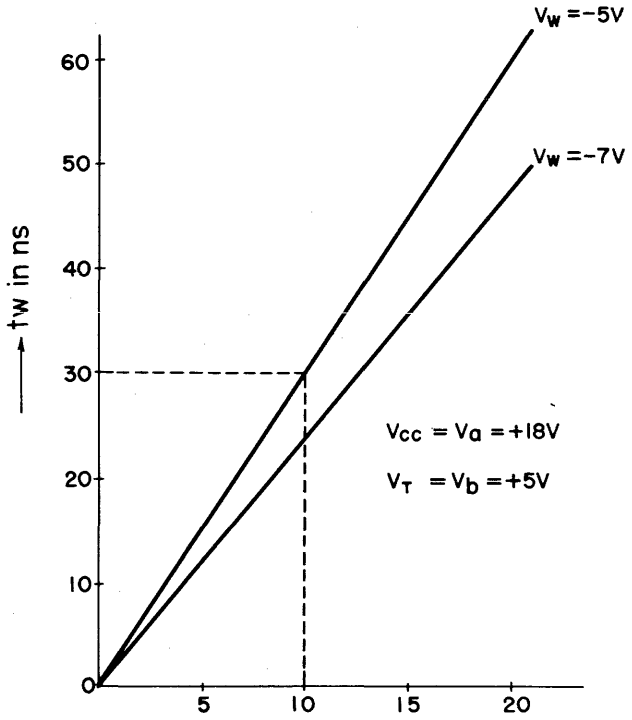


Figure 4 – Switching characteristics of the associative memory

and a mismatch signal is generated as shown in trace (f). Then the cell is interrogated by the "1" interrogate pulse of trace (e). There is no mismatch signal found in trace (f). The "0" write digit pulse of trace (c) and the third word pulse are used to change the cell from the "1" state to a "0". The cell is again interrogated by "1" and "0" interrogate pulses. Figure 5 shows a sequence of the above operations.

To improve the characteristics in the read operation after the write operation, some investigation is being performed. One of the methods is that Q7 and Q8 are used for interrogate and read operations, while Q5 and Q6 are used for write operation only. With this modification, a certain improvement would be expected.

Associative memory system

As mentioned already, the integrated MOS transistor associative cell has low power dissipation characteristics in interrogate operation. Moreover, since word matching signals can be obtained from associative memory matrix during the entire interrogate operation, the interrogate operation and sequential read operation of matching words are simultaneously performed until the entire multimatch resolving process is finished. Therefore, multimatch resolving can be done in high-speed by use of integrated MOS transistor associative cells and high speed peripheral circuits. Of course the most important function is to attain a high speed operation, low cost and simple multimatch resolver.

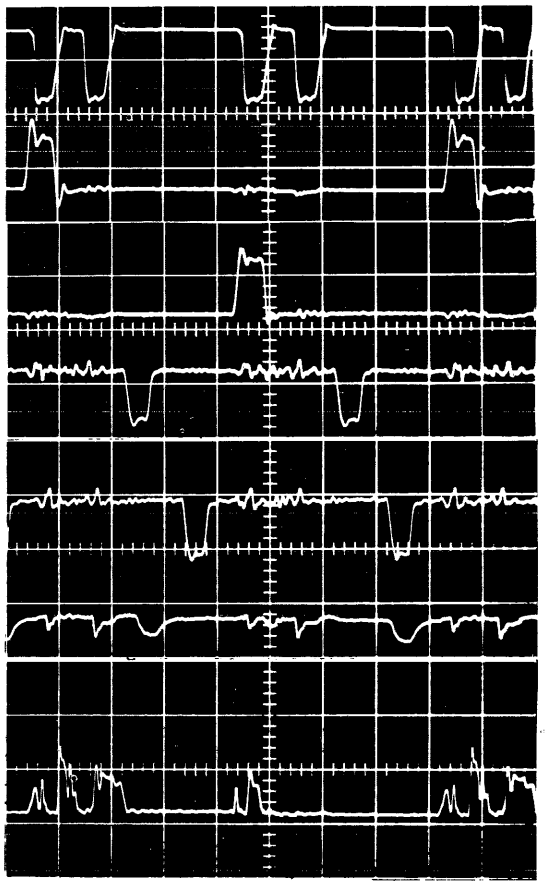


Figure 5 – Results of measurement

Although some methods have been reported about multimatch resolving,[3,4] this paper describes a different approach in the multimatch resolving.

As shown in Figure 6, general sequence of events taken place is that firstly an arbitrary interrogate information activates interrogate drivers and the interrogate information is compared with the stored information in an associative memory matrix. Signals indicating matching words are fed into the detector matrix for dividing X and Y components. These components are stored in the X and Y-multimatch resolvers and the priority is given to each resolver. Based on X and Y configurations, a specific word driver is activated for the selection. The process continues under the priority control until X and Y-singlematch detectors detect the end of current interrogation.

The details in our system will be described below with an example.

The associative memory matrix consists of 128 words of 48 bits per word with associative memory cells which are described in section 1. This memory matrix provides the matching signals on the word-sense lines corresponding to matching words to the detector matrix after receiving interrogate information.

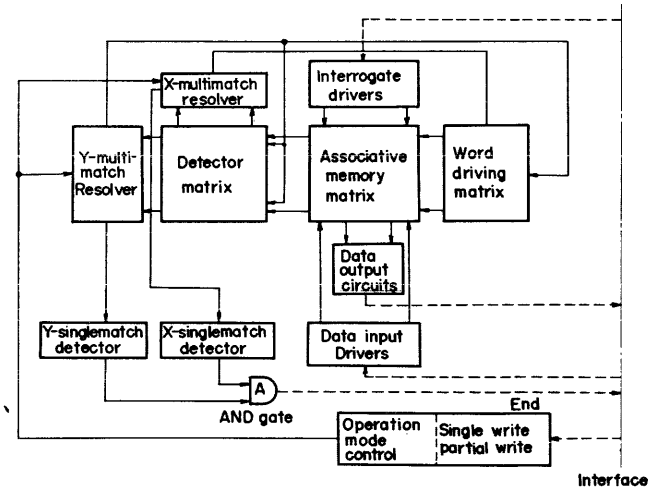


Figure 6 – General block diagram of MOS transistor associative memory

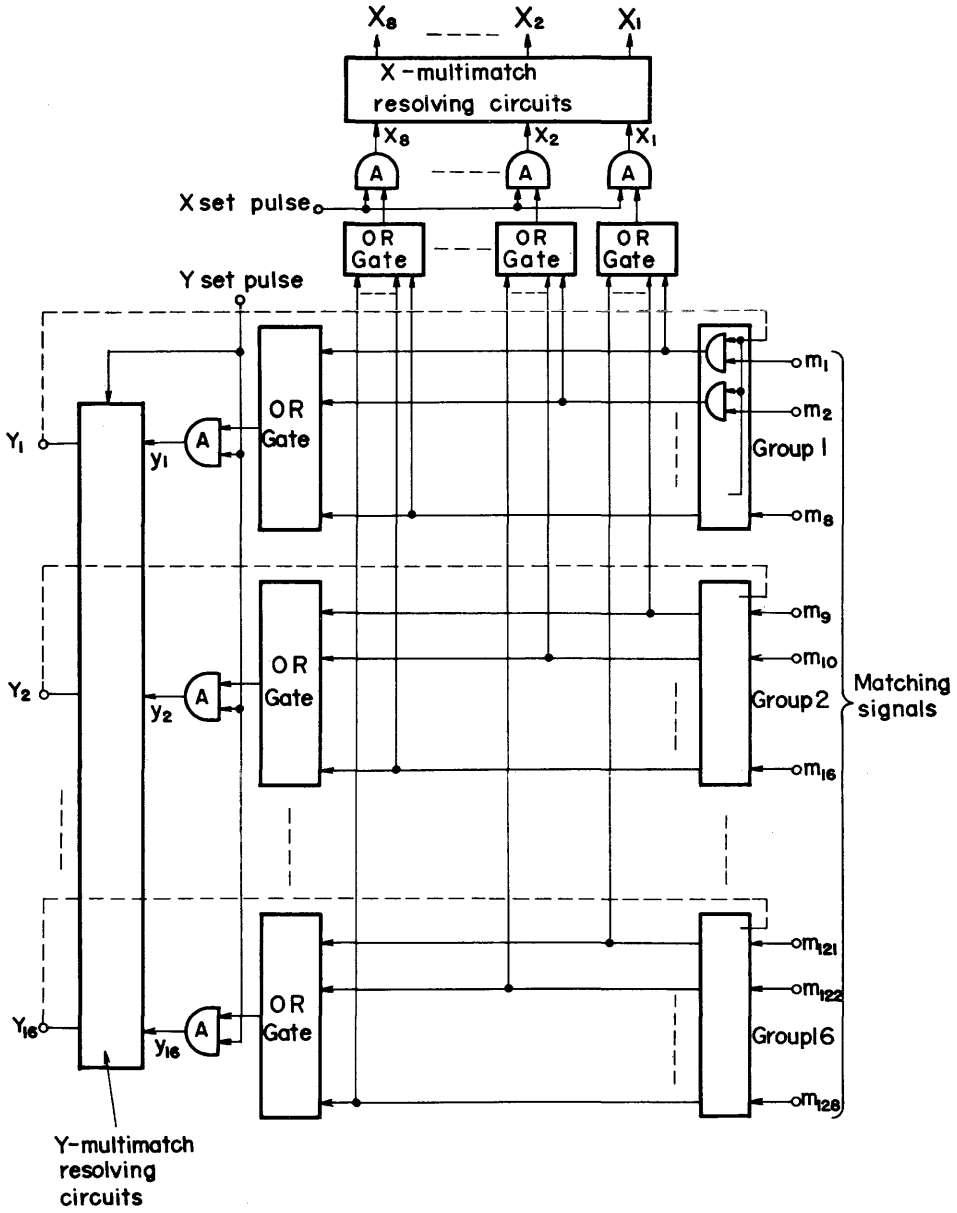


Figure 7 – Logical diagram of detector matrix

The detector matrix shown in Figure 7 has an essential function for multimatch resolving. The matching signals generated by each word-sense line of 128 words are given to the 128 AND gates which are divided into 16 groups as shown in Figure 7. Each group has 8 AND gates.

Logical diagrams of the Y-multimatch resolving circuits and X-multimatch resolving circuits are shown in Figure 8 and 9 respectively.

Initially, flip-flop FY-1 in Figure 8 will be set by some of word matching signals m_1 through m_8 in the group 1 when Y set pulse shown in Figure 8 is enabled. Also at the same time flip-flops FY-2 through FY-16 will be set by some of the word matching signals in the group 2 through 16 respectively. Thus set state of FY-1 through FY-16 shows that at least one matching word exists in the corresponding group. The flip-flops FY-1 through FY-16 in Figure 8 and the flip-flops FX-1 through FX-8 in Figure 9 are modified J-K flip-flops and consist of CML integrated circuits as shown in Figure 8. This flip-flop has the feature that it is set at the front edge of a set pulse given to S terminal and is reset at the trailing edge of a reset pulse given to R terminal. Y control shown in Figure 8 is normally in "1" state. Thus the priority in the order of Y_1, Y_2, Y_3, ..., Y_{16} is established by the Y control.

It is assumed that, for example, at least one of matching signals in the group 2 and the group 11 are provided, and then FY-2 and FY-11 are set, but only

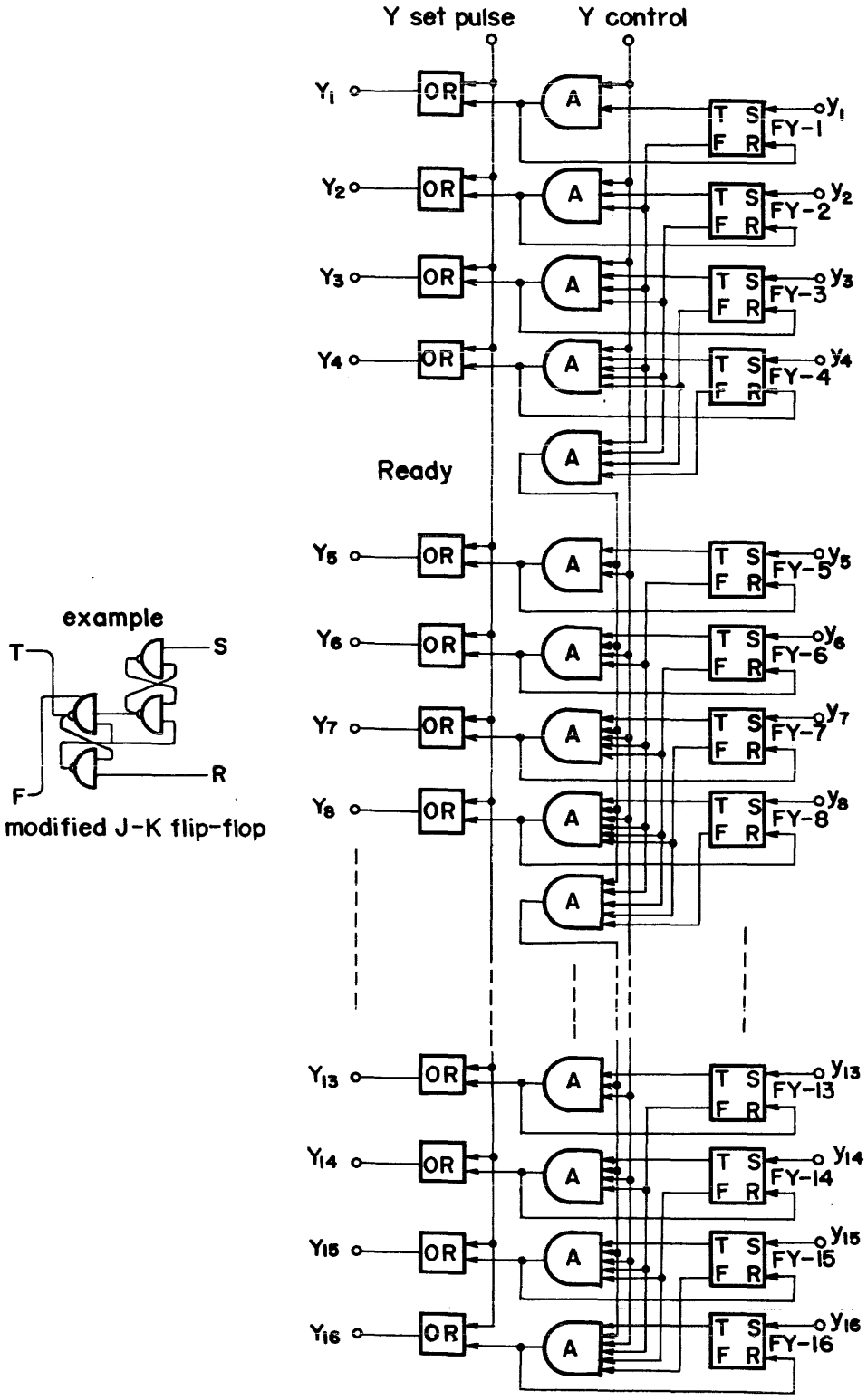


Figure 8 – Logical diagram of Y-multimatch resolver

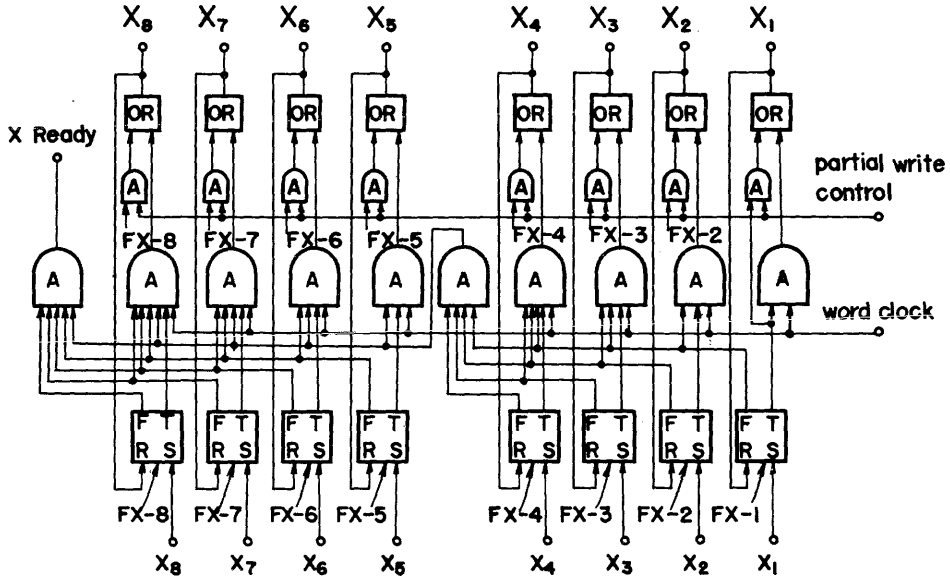


Figure 9 – Logical diagram of X-multimatch resolver

Y_2 is in "1" state for processing the matching signals m_9 through m_{16}. When the entire processing in the group 2 is over, then Y control temporarily becomes "0" state to reset FY-2 and the priority control moves down to FY-11. Therefore, for the above example, when Y_2 is in "1" state, the states of m_9 through m_{16} are stored in flip-flops FX-1 through FX-8 respectively. The states of 8 matching signals in the one group with the highest priority will be stored in FX-1 through FX-8 by enabling X set pulse as shown in Figure 9. Suppose FX-3 and FX-8 are set by m_{11} and m_{16}. Thus the states of FX-3 and FX-8 show the matching words in the group 2. The word clock in Figure 9 establishes the priority in the order of $X_1, X_2, X_3, \ldots, X_8$. The overall timing chart for the above example is shown in Figure 10 and the first word clock comes, and then X_3 as the first priority in Figure 9 becomes active. Then the first driving signal is generated by Y_2 and X_3 in the word driving matrix. Therefore the first matching word is selected. After the first word clock is over, FX-3 is reset and the priority moves down to FX-8 enabling X8. When the second word clock comes, the word driving signal is generated by Y_2 and X_3 in the word driving matrix to select the second matching words in the group 2. When the selection of all the matching words in the group 2 is finished, Y control reset Y_2 and enables next higher priority Y signal such as Y_{11}.

Now the states of matching signals m_{81} through m_{88} are stored in FX-1 through FX-8. Under the X priority control, the selection for the matching words of m_{81} through m_{88} will be performed. The selection process continues in the same manner under the X and Y priority controls until X-singlematch and Y-singlematch detectors detect end of the process. The logical diagram of Y-singlematch detector is shown in Figure 11. The X and Y-singlematch detectors are so designed to give a signal to the central processor for the preparation of the next action one word clock prior to the actual last word clock to speed up the interrogate repetition.

When the partial write operation is desired, the partial write control becomes active and in effect gives the function to by-pass the priority, therefore

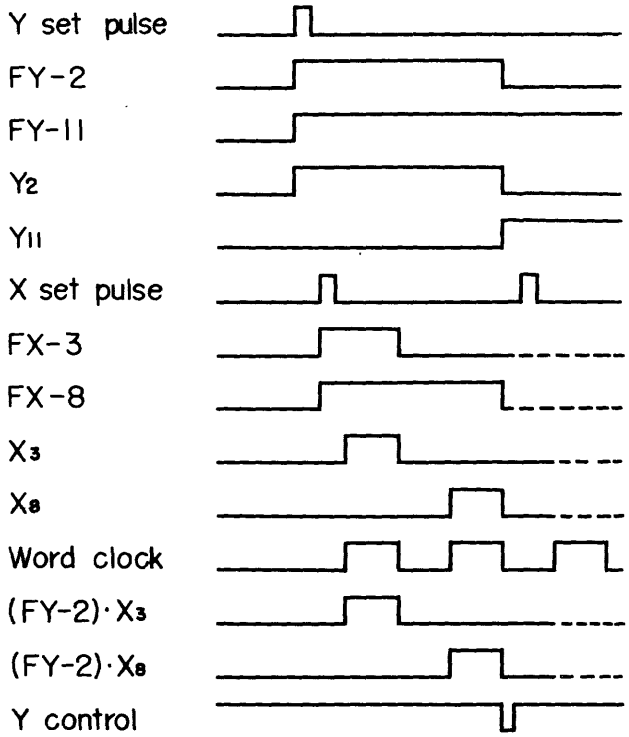


Figure 10 – The timing chart

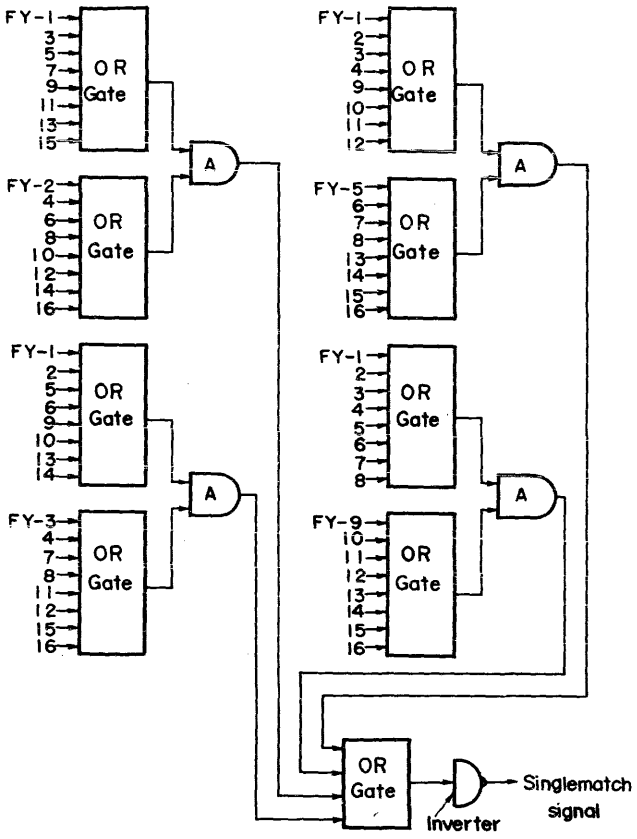


Figure 11 – Logical diagram of Y-singlematch detector

X_1 through X_8 become simultaneously active depending on the states of FX-1 through FX-8.

CONCLUSIONS

The low-power requirements of the associative cell have made it possible to build an integrated MOS transistor associative memory system.

MOS devices will be useful in large arrays of memory applications where fairly high-speed operation is required.

Four bits of associative memory on a single chip have been fabricated. However, it is expected to have a larger quantity of associative memory cells on a chip with MOS transistor circuits.

It is found that by decreasing the amplitude of interrogate drive pulses and increasing the mutual conductance of MOS transistors used for interrogation, the power dissipation in interrogate operation will be decreased and the signal-to-noise ratio on the word-sense line will be increased.

The method of multimatch resolving technique described in this paper is quite effective to realize a simple high-speed circuit system and an easy control of the partial write and single write operations.

ACKNOWLEDGMENTS

The authors wish to acknowledge K. Nagamori and S. Tsuneki for their constant encouragement and advice, and T. Kurosawa, T. Ishidate, M. Shiraishi and H. Yamamoto for valuable help during discussions.

REFERENCES

1 R IGARASHI T KUROSAWA T YAITA
A 150-nanosecond associative memory using integrated MOS transistors
ISSCC DIGEST of TECH. PAPERS February 1966

2 HAROLD BORKAN PAUL K WEIMER
An analysis of the characteristics of insulated-gate thin-film transistors
RCA Review June 1963

3 J R KISEDA H E PETERSEN W C SEELBACH M TEIG
A magnetic associative memory
IBM Journal April 1961

4 EDWIN S LEE
Associative techniques with complementing flip-flops
Spring Joint Computer Conference Proceedings 1963

Plated wire bit steering for logic and storage*

by W. F. CHOW**
Bell Telephone Laboratories
Murray Hill, New Jersey

and

L. M. SPANDORFER
Univac Division of Sperry Rand Corporation
Blue Bell, Pennsylvania

*Portions of the work described in this paper were supported under Contract DA28-043 AMC-01305 (E) between the U. S. Army Electronics Command, Fort Monmouth, New Jersey and the Univac Division of the Sperry Rand Corporation, Blue Bell, Pennsylvania.

**Dr. Chow's contributions to this work were made when he was with Univac Division of Sperry Rand Corporation, Blue Bell, Pennsylvania.

INTRODUCTION

This paper describes a new form of magnetic logic technology known as bit steering which utilizes plated wire for both storage and logic and permits logic manipulations directly within the memory matrix. Bit steering exploits the inherent plated wire NDRO property, the excellent coupling between the wire and its magnetic surface, the relatively thick magnetic film, the low impedance possibilities of the wire loop, the stripline transverse strap loop, and the continuous fabrication process which provides uniform properties along the wire.

Two of the limitations to widespread application of earlier magnetic logic elements have been relatively slow speed, typified by simple ferrite structures which generally use domain wall motion, and the use of discrete elements with complex winding and interconnection patterns. In contrast, plated wire provides a fast rotational switch permitting bit transfers in excess of 3 MHZ, and a reasonable level of batch fabrication without complex interconnections.

In view of the current advances and pervasiveness of the semiconductor large scale integration (LSI) concept, this paper will focus on structures for parallel processing applications in which bit steering may have significant advantages over semiconductors such as large content addressable memories (CAMs), long first-in first-out (FIFO) buffer registers, and new random access memory structures. Bit steering, for example, will be shown to provide a method of drastically reducing the cost of word logic, heretofore the dominant cost element in magnetic CAMs.

Earlier forms and implementations of bit steering have been described by other workers.[1-4] The present approach is most closely related to recent work reported by Dick and Doughty,[5] and Dick and Farmer,[6] with the main point of departure being the system exploitation of NDRO operation.

The plated wire memory element

The plated wire used in bit steering is essentially the same as that used in conventional memory arrays and has been described by numerous workers.[7-11] Briefly, it is fabricated by continuous electroplating of an approximately one-micron thick permalloy film upon a typically 5-mil diameter copper beryllium wire. A memory matrix is formed by means of a simple orthogonal arrangement of plated wires and an overlay of copper straps. The intersection of a plated wire and a strap, shown in Figure 1, constitutes a memory element.

In the conventional or voltage mode, the memory is organized as shown in Figure 2. One or more

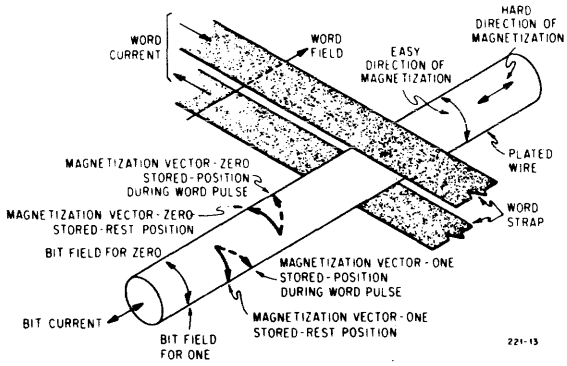


Figure 1 – Storage element organization

words lie under one strap; low level switches can be used for connecting the wires to the sense amplifiers and to the wire drivers. The block on the left side of the diagram contains the strap drivers and strap select logic; the block on the bottom contains the wire drivers, the low level switches, the wire select logic, and the sense amplifiers.

The basic NDRO switching speed of the plated wire element depends upon the rise time and level of the applied currents. Typical switching speeds of 70 nanoseconds are obtained with strap current rise times of 30 nanoseconds, and switching times of 7 nanoseconds can be obtained with fast rise time, high level pulses operated in a time-limited mode. Conventional memories of thousands of words per matrix appear feasible with a read cycle time of 100 nanoseconds or less.

Bit steering mode

Plated wire requires coincidence of a wire current with a strap current for writing; the conventional source of the wire current is an external circuit. When reading in the voltage mode, the current flowing in the wire is small and essentially the full bipolar sense voltage appears at the sense amplifier input. In the bit steering or current mode, the fundamental idea is to use the sense voltage generated in the magnetic element itself to serve as the source of wire current and selectively write or transfer into another element on the same wire. To enable the switching voltage (≈30-60 millivolts) to generate sufficient wire current to exceed the write threshold (≈10-35 milliamperes) it is necessary that the plated wire loop provide a transmission impedance of the order of several ohms. This requires elimination of all conventional series impedances in the loop such as the input impedance of a sense amplifier.

Since the loop is essentially a short circuited transmission line, it can be represented by an approximate equivalent circuit of a lumped inductance L in series with a resistance R as shown in Figure 3. The inductance is dependent on factors such as the loop geometry, the magnetic properties of the plated wire, and current rate of change. One loop structure used for positioning the wire and achieving low inductance is called the seimicoaxial and is shown in Figure 4. The loop has a complex cross-sectional structure whose characteristics are difficult to calculate. As an approximation, however, the inductance of an ideal full coaxial can be expressed as

$$L = \frac{\mu_0 t}{\ell} + \frac{\mu t_p}{\ell} + \frac{1}{\ell}\sqrt{\frac{\mu_0}{4\pi\sigma f}}$$

The first term on the right is the loop self inductance assuming zero penetration into metal, where μ_0 and t are the permeability and thickness (≈0.3 mil) of the dielectric, and ℓ is the perimeter. The second term on the right is the component of inductance due to the permalloy, where μ ($\mu_r \approx 50$) and t_2 are the permeability and thickness, respectively of the film. The third term actually consists of two components, one due to penetration into the wire and one due to peretration into the copper ground plane; σ is the conductivity and f is the frequency. At 25 MHZ, approximate values for the four components of inductance are 0.6, 4, 1, and 0.5 nanohenries per inch, respectively. With the close spacing, the component due the permalloy dominates the loop inductance. For plated wire currently being made and used for conventional memories, the measured and computed values of the loop inductance for various forms of experimental coaxials are in satisfactory agreement and range from about 6 nanohenries per inch for the ideal full coaxial to 14 nanohenries per inch for simpler, practical configurations. The resistance R including skin effect of a loop consisting of the plated wire and a copper ground plane is about 0.2 ohms per inch. Variations in R and L for loops of the order of 6 inches can be partly masked by the inclusion of a small current-controlling resistance R_s of one ohm or less.

Techniques for obtaining nominal reduction of the inductance include the use of segmented rather than

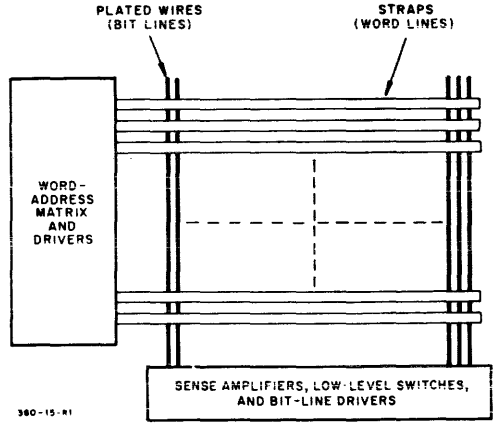


Figure 2 – Block diagram of conventional memory

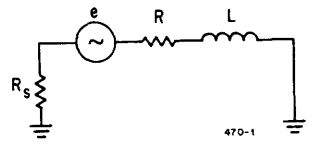


Figure 3 – Loop equivalent circuit

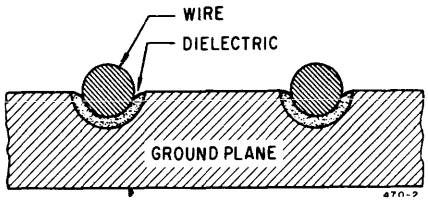


Figure 4 – Semi-coaxial structure

continuous films and the use of thinner dielectrics. The effective loop resistance could be decreased by about 30% with the use of copper rather than copper beryllium. One of the problems in achieving the low inductance structure in practice arises in the need for mounting and terminating the wire onto the semi-coaxial structure without introducing skew. Deposited magnetic film closed-loop structures can potentially provide a reduction in the necessary low inductance and resistance at the same time bypass the problem of mounting and terminating the wire.

Gain

Two fundamental problems in all-magnetic logic pertain to methods of achieving directionality and gain. Directionality in the present form of bit steering is handled largely by the availability of excess gain and by appropriate selection of straps under control of external circuits. In the past, gain has often been obtained by techniques such as turns-ratios in transfer windings, multihole structures, and use of the fall-back mode. The fall-back mode, i.e. coincidence of steering current with removal of the receiver hard axis field, has been exploited for both DRO shift register operation[6] and general purpose DRO logic. Maximum gain is achieved in the fall-back mode when either (or both) the source and load bits are driven destructively (and simultaneously) by large hard axis fields ($\approx 2H_k$). In NDRO applications, however, only a limited amount of fall-back can ordinarily be utilized, hence NDRO operation inherently provides less gain than DRO. This apparent disadvantage can be offset by noting that many parallel processing structures do not require a large amount of gain.

In a bit serial CAM, for example, it can be shown that the transfer requirements within the matrix for the major operations such as search, write-on-match, resolve, write, etc. can be conveniently satisfied by always combining two elements as a double strength source or transmitter to drive one element as the receiver, The logical gain of one-half can be exploited with a negligible increase in hardware and without incurring a trade-off on speed.

If $\Delta\phi_1$ is the amount of flux change produced by the strap currents, $\Delta\phi_2$ the flux change at the receiver, $\Delta\phi_3$ and $\Delta\phi_4$ the flux change required to overcome the loop resistance and inductance respectively, the condition $\Delta\phi_1 = \Delta\phi_2 + \Delta\phi_3 + \Delta\phi_4$ is required for transfer. For a pair of combined sources, each source must ideally supply $\Delta\phi_1/2$ units. Figure 5 shows the magnetic vector relations in an idealized model. The load or receiver element vector is assumed driven through angle α_1 by the strap current. Assuming equal sources and load strap currents, α_1 is also the initial angle of rotation of each source. The load is then tipped through angle α_2 (if the sources are the same polarity and opposite the load) at the termination of the source strap current pulse. Assuming limited fall-back is not used, the load element finally rotates through angle α_3 upon removal of the load strap current pulse. At the source angle of 60° the theoretical maximum angle of rotation of the load is 60°, under the assumption of a pair of elements for the source. Under this limiting condition the loop resistance and inductance must be zero. Thus transfer requires angles in excess of 60°; it is clearly desirable to minimize tolerances on parameters such as H_k and dispersion to permit as large a source angle as possible without exceeding the NDRO limit, and to use limited fall-back if needed. The gain enhancement possible with the so-called limited fall-back mode is indicated in Figure 6. The variable phasing indicated in Figure 6a provides a varying amount of switched output from an element as shown in Figure 6b.

Several phasing arrangements which have been used in transfers are shown in Figure 7. Elements A and B serve as the transmitter pair and F serves as the receiver. In Figure 7 (b), if I_f overlaps I_A and I_B, a bipolar transfer results in the complement of A and B (assuming they are in the same state) finally being stored in F. Ideally no steering current flows if A and B are in opposite states, resulting in F remaining unchanged. The bipolar signal insures a history balanced in 1's and 0's since no more than

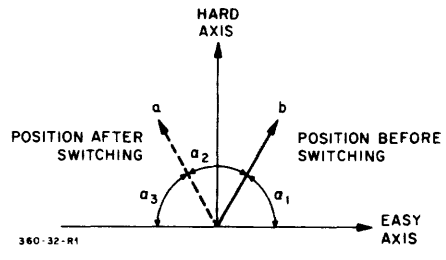
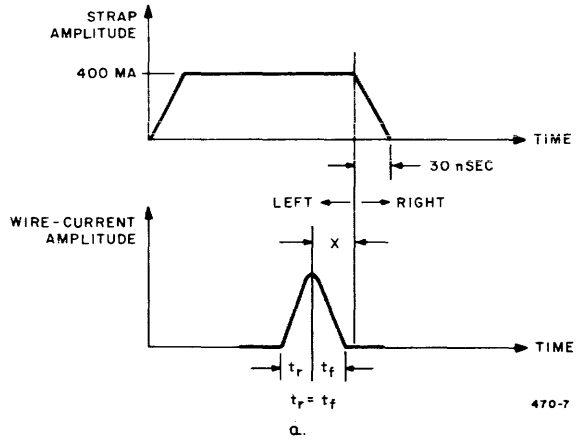
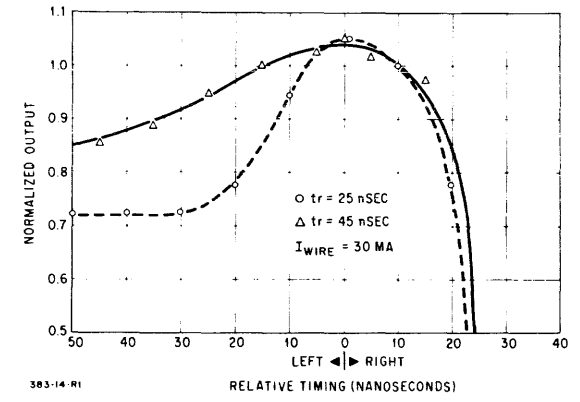


Figure 5 – Idealized vector relations



a.



b.

Figure 6 – Limited fall-back mode: (a) timing relations, (b) gain enhancement

two like bits can be written consecutively in the same element. This mode tends to minimize the possibility of crawl due to imperfect cancellation and interference from adjacent elements. The limited fall-back mode is shown in Figure 7 (c); the trailing edge provides a post-write disturb which, like the bipolar mode, tends to prevent spreading.

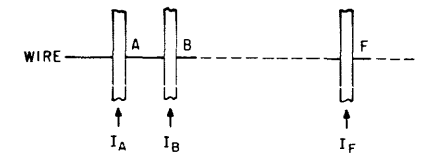
An example of a sequence of fall-back mode transfers is shown in Figure 8a; a wire requiring 25 milliamperes write current was used. The sequence begins with a transfer from a pair of source bits containing binary 1's into a receiver on the first clock pulse. The receiver is then sensed by three successive transfers. On the fifth clock pulse, a pair of source bits containing 0's transfers into the receiver, thus completing the cycle. Note the constancy of amplitude of the repeated read. An enlargement of one of the transfer signals from the receiver is shown in Figure 8b. The two inch wire used in this transfer was lying loosely on a flat ground plane providing roughly 20 nanohenries per inch. The decay time constant of about 100 nanosecond is in satisfactory agreement with a simplified model consisting of a triangular voltage waveshape in the wire and the assumed values of L and R including a 0.33 ohm series viewing resistor. The element was formed by three turns of a 0.035 mil-wide strap; a strap current of 600 milliamperes was used with a rise time of 10 nanoseconds. Multiloop array structures which inherently make use of short loops of the order of several inches or less are described in subsequent paragraphs. Maximum length transfer experiments have been performed on 7-10 inch loops with 20 milliampere wire and a strap density of 8 bits per inch.

Hard axis read

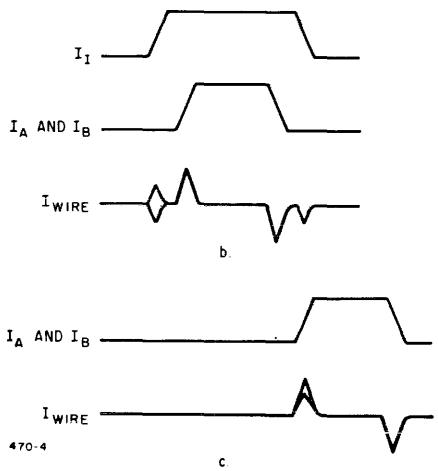
In many applications the most useful read-out-direction is orthogonal to the word. In addition, since the wire loop is practically short-circuited at both ends, the conventional readout method in the direction of the wire cannot readily be used. Read operation is therefore based on the change of open-loop flux in the hard-axis direction.

Under static conditions, the orthogonality of the plated wire and straps insures that coupling does not exist between a wire current and the strap. To provide coupling, bias current is applied to the straps as shown for a balanced strap pair in Figure 9 (a). This current rotates the vectors away from the easy axis, enabling flux linkage with the straps. The current in the plated wire will then modify the angle away from the easy axis and consequently a signal is induced on the straps. The phase of the readout signal induced on a biased balanced strap pair depends on the polarity stored at the element as shown in the waveforms of Figure 9 (b) and (c).

The flow of wire delta currents due to imperfect cancellation of opposite-state source elements in



a.



b. c

Figure 7 – Transfer phasing arrangements

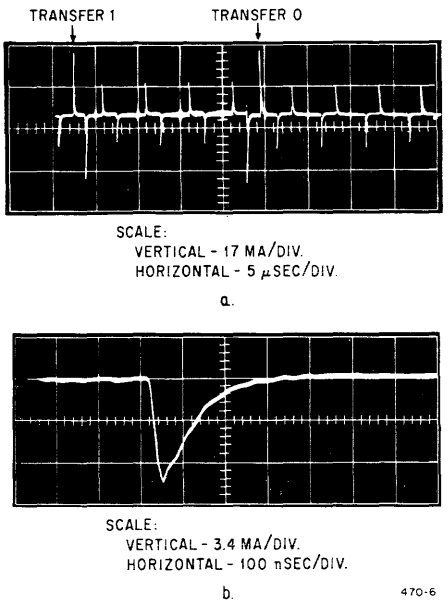


Figure 8 – Transfer sequence: (a) transfer 1, readout 3 times, transfer 0, readout 3 times, (b) enlargement of readout signal

unselected wires of an array introduces noise into a biased strap. This noise source appears to place an upper limit on the number of series in a uniform array module. (A uniform array is one in which all straps intersect all wires.) Extrapolations from experiments on 12-wire arrays give indication that the upper limit on uniform arrays is about 256 wires; large scale noise tests will be required before maximum module size can be firmly stated.

The bit-steered array

The general structure of the bit-steered array developed thus far will be reviewed. Figure 10 shows such an array; the only connection to the external circuits is through the straps. It is important to observe the electronics block connected to the plated wires in Figure 2 does not exist in the bit-steered array; thus control can be achieved from one periphery. It will be shown that any desired plated wire can be randomly accessed for search, write, read, or any other function by merely sequencing the drive currents in the appropriate set of straps.

Since the array is controllable from one periphery, the electronics cost to a first approximation is proportional to the length of that periphery. Thus a system with relatively few straps and a large number of wires is expected to have minimum electronics cost, subject to design constraints such as the delta noise read problem and the requirements on the high current strap drivers. The back voltage spike which occurs upon application of the strap current in, say, a 256 wire bit steered module with wires spaced on 30 mil centers ranges from 6 to 10 volts depending on whether a double or single-sided ground plane is used.

It is possible to fabricate the wire substrate and the strap overlay as separable parts, thus providing the basis for a removable media solid state memory; this feature has already proved to be of considerable convenience in laboratory development work.

Basic operations

The bipolar phasing shown in Figure 7b is naturally complementary. The Nand function $F = \overline{AB}$ can be executed by first setting F to 1 with a pair of reference straps each containing a permanent 1, and on the second step combining A and B as a pair and transferring into F. If A and B both contain 1, a 0 will be transferred; otherwise F will remain at 1. Similarly Nor can be executed by first setting F to 0. The And and Or can be executed with the non-complementary transfer shown in Figure 7c.

A useful notation for transfer is (X, Y, Z), where X and Y denote the transmitter and Z denotes the receiver. Thus the two-step sequence for $F\ F = \overline{AB}$ is

(1) (1, 1, F)
(2) (A, B, F)

In a CAM, the information bit d can be stored in either single or two rail form; two-rail operation uses a pair of elements A and B (see Figure 11) for the data. If d = 1, a 1, 0 is stored at A, B; if d = 0, 0, 1 is stored. F is initially set to 0. To search d for a

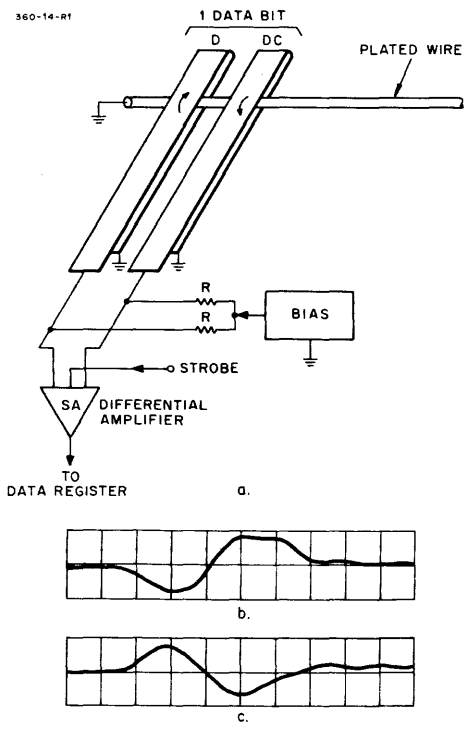


Figure 9 – Hard axis readout: (a) organization, (b) sense voltage: 1, (c) sense voltage: 0

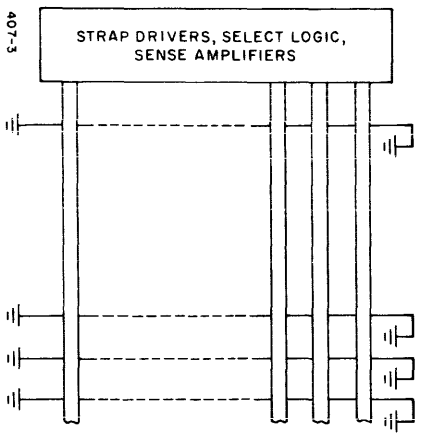


Figure 10 – Bit steered array organization

binary 1, (R_1, B, F) is performed where $R_1 = 1$; to search for 0, (R_1, A, F) is performed.

After all data bits are searched, the status of the flag F indicates the response of the word. If F still contains a 0, a match will be indicated; if F has switched over to a 1, a mismatch will have occurred. It should be noted that once a flag switches to 1, it cannot inadvertently switch back to 0 as the search proceeds through the remainder of the data field.

Single-rail operation is similar to two-rail but uses two reference straps R_1 and R_0 ($R_0 = 0$) for reference 1 and 0, respectively, and two flag straps F_1 and F_0 (not shown). A selection of appropriate reference and flag straps is required on each bit search. At the end of the word search, F_1 and F_0 can supply a limited amount of information on greater-than or less-than search. To achieve common mode cancellation for readout, it is desirable to distribute several dummy straps throughout the array.

When the search operation is complete, it is useful to store the complement of flag F in flag F′ (not shown). The write-on-match operation can then be executed by transferring from the F=0 or F′=1 flags back into the data field. This is done by pulsing F in conjunction with R_0 to write a 0, or F′ and R_1 to write a 1; all wires that have satisfied the match criteria will thus receive steering current.

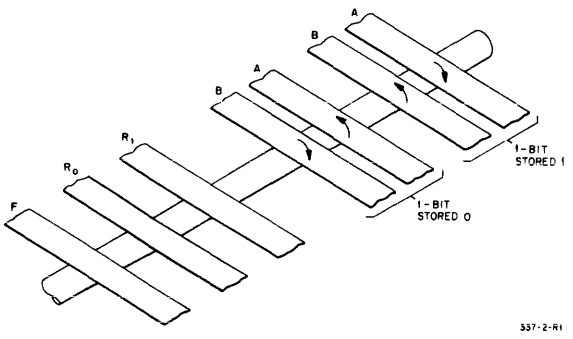


Figure 11 – CAM storage elements

The conventional read or write operation is performed in a similar manner; the basic difference is the need to select or address one specific wire. This can be done by means of an address or decoding field which contains a fixed pattern consisting of the natural binary sequence from 1 to N corresponding to the N wires in the array. Figure 12 shows an illustrative 8 word single rail system. In general, $\log_2 N$ straps are required. The address in placed in the address register whereupon a conventional search is initiated across the address field. At the end of search, one and only one match can exist. As indicated, a 101 in the address register will result in all flag bits other than that on wire 6 being set to 1. Upon the termination of the address cycle, the flag strap along with R_0 can be used on a succeeding read or write cycle to activate or mark the selected wire.

Ordered retrieval of multiple matches

The logic flexibility of the bit-steered structure permits many of the well-known retreival algorithms to be executed; the criteria may be based on data content or on physical address. One requirement might be to resolve multiple matches, one at a time, in ascending or descending address order.

The resolve structure for such a retrieval makes use of the address field already described and in addition, requires several temporary register straps. An algorithm for manipulating this set of straps to provide a binary decision or binary sieving action has been described[12] which permits resolution of each match in approximately $4 + 21\log_2 N$ basic transfer cycles, where N is the number of words in the system.

The external control logic required to execute the resolve algorithm is minimal and consists of several short ring counters, Nand gates, and flip-flop registers in addition to the strap drivers and one sense amplifier.

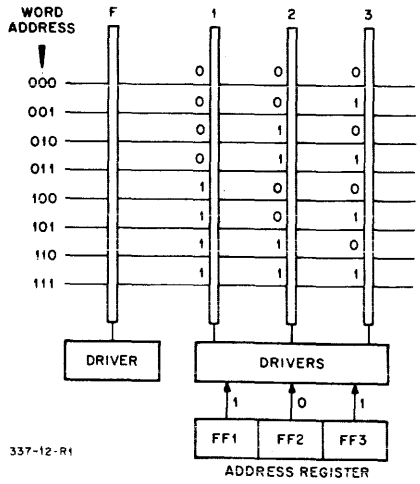


Figure 12 – Address field

The multiloop structure

The bit-steered array described thus far has a basic limitation on wire or word length due to the linear increase in loop inductance and resistance with loop length. In an effort to provide a structure which removes the limitation on word length, permits wider tolerances on plated wire, and at the same time increases the search speed from bit-serial to quasi bit-parallel, the multiloop configuration shown in Figure 13 has been devised. It basically provides a method in which adjacent loops are decoupled as far as the L, R transmission parameters are concerned, but nevertheless permits bit transfer between elements in adjacent loops. An NDRO version of the basic Dick-Farmer topology[6] could serve a similar function. Figure 13 (a) shows a simple structure with two loops 1 and 2. The short section of plated wire labeled A permits information to be transferred from the bit under strap W to that under strap X. Due to the low impedance from point P to ground via A, the steering current generated by W flows mainly into A, and only a small leakage current flows into loop 2. The length or impedance of the short to the long section is about 6-10:1 or larger. The relatively small values of L and R permits the insertion of current-controlling resistances into each loop (not shown). On the next transfer cycle it is possible to transfer from the bit under X to the bit under Y. The approximate 50% loss of current back into loop 1 can either be tolerated or partially bucked out, if necessary. Simultaneous transfer can be executed along the two long wire segments without mutual interference, providing neither is transferring to X.

The two loops can be used to divide a CAM, for example, into two convenient sub-loops, namely one for the data field, and one for the address and resolve fields. This subdivision permits search or write operations to proceed simultaneously with addressing or resolving, in addition to reducing L and R.

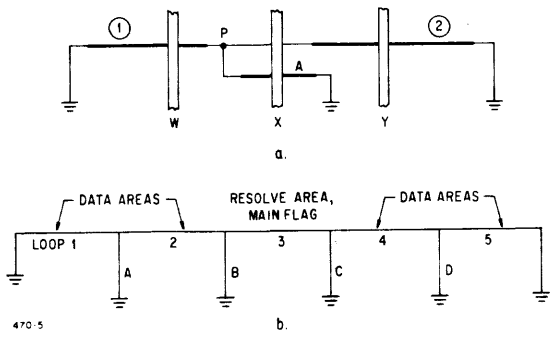


Figure 13 – Multiloop configuration: (a) basic two-loop structure, (b) extended structure

Figure 13 (b) shows an example of an extension of the two-loop scheme into five loops which might comprise one CAM word. The five plated wire loops are numbered 1 to 5. The four short lengths of plated wire are labeled A, B, C, and D. Wire A is common to loops 1 and 2 and wires B, C, and D are common to loops 2 and 3, 3 and 4, and 4 and 5, respectively. Sections 1, 2, 4, and 5 might contain the actual data, and section 3 the address tree and other straps required for resolve. The effective word length is the sum of the bits of the four sections.

In each section auxiliary flags store the search result of the individual data fields. The auxiliary flag data can be logically combined by a shifting process into the main flag in section 3; it indicates the match status of its corresponding word and also plays the usual role in the random-access operation. The auxiliary flags store the search result within each data field and serve as the source of loop current for the read/write operation within its data field.

When a shifting search is executed in the multiloop structure, a significant improvement in search speed is possible. As an example, assume there are four loops each of 13 bits and that it is desired to search 7 of these bits in each of the four fields. In a single loop this operation requires 28 transfers; but in multiloop only 7 transfers are required for search and 3 for consolidation of results for a total of 10 transfers. This represents a speed improvement ratio of almost 3 to 1.

In general if there are n bits per word, the total search time is $\left(\frac{3\sqrt{n}+}{2}\right)$ transfers. The improvement in speed becomes greater as the number of data fields and the number of searched bits per data field increase. The improvement is also realized in the write-on-match and write operations.

Scanned memories

The multiloop technique can also be used for scanning or pointing. The basic pointer is shown in

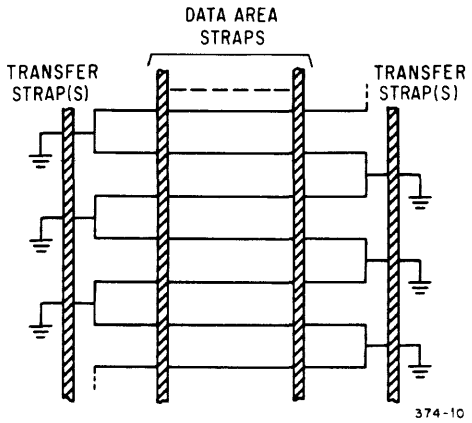


Figure 14 – Pointer structure

Figure 14; it is essentially the same structure shown in Figure 13 with the exception that the colinear sections are folded, accordion style, and stacked as ladder rungs. This makes it possible to transmit, for example, a single binary 1 in a zigzag fashion up or down the structure.

In addition to pointing, patterns of 1's and 0's can be shifted up or down the ladder as in a shift register. The data area and the shift feature in the structure of Figure 14 provides the basis for a block oriented memory and for a FIFO buffer. In the former, the data blocks can be organized vertically. Access time to a vertical block requires one steering cycle to reset the pointer, wherever it is located, and then a number of cycles to set the pointer at the head of the block which depends on the system organization. Once a block is accessed, the bit serial rate equals the pointer rate. It is possible to operate with only a single sense amplifier which can be connected to a special readout strap. Since the array data is always stationary in this system (only the pointer moves), a sudden loss of power during the operation will not result in data loss.

In the FIFO application, a horizontal data organization can be used as shown in Figure 15. The character is oriented along the wire. All characters are shifted up one rung when each new character is entered. The pointer can be used for marking the first-in character; it is used in conjuntion with a reference strap to locate and readout the first-in character. The pointer is then stepped to the next lower rung where it marks the new first-in character. It is also possible to operate the FIFO with a single sense amplifier.

A random addressed rather than a scanned memory can be devised which does not use a pointer but which instead consists simply of a data field and an address field of the type shown in Figure 12. If the data width

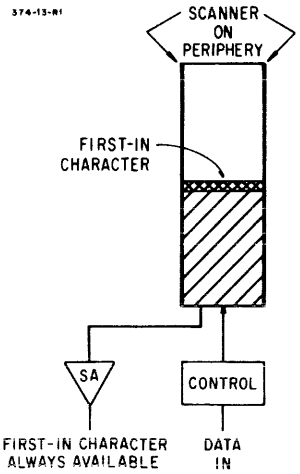


Figure 15 – First-in first-out buffer

is limited to several bytes, one loop of 35 milliampere wire may be sufficient. The number of transfer cycles required for addressing is $\log_2 N$ plus several housekeeping transfers, where N is the number of words in the array. The addressing speed can be improved by using a two-loop structure which permits readout of one byte to proceed simultaneously with the addressing of the next byte. It is possible to further increase the addressing speed by the use of address trees with codes other than the natural binary code used in Figure 12. One possibility is a 2-out-of-k code; for example, a 1024 wire system can be addressed with two slipped 9-strap fields (k = 9) in three transfer cycles. This compares with 10 cycles and 10 straps with the binary code. Conversion from a natural binary machine code to a non-algebraic 2-out-of-k is awkward and limits the technique to special applications.

Complex logic

The basic logic operations were described earlier. More complex functions can be handled by a technique reminiscent of cellular logic. An illustrative function, using non-complementary transfers, will be described:

$$f = x_1x_2x_3\bar{x}_4 + x_1x_2\bar{x}_3 + \bar{x}_1x_2 + \overline{x_2}\bar{x}_3x_4$$

There are several ways to set up such a function in a bit steered array; the pattern of 1's and 0's in Figure 16 indicates one possibility. The variables x_1 through x_4 enter the cellular array in two-rail input lines. Each plated wire is assigned to one of the four prime implicants; $x_1x_2x_3x_4$ will be formed on plated wire 1, and so forth. Since x_1 appears uncomplemented in the first two prime implicants, binary 1's are stored on the strap pair assigned to x_1 on wires 1 and 2. Since x_1 is complemented in the third prime implicant, a 0 is stored on wire 3. Since x_1 does not appear in the fourth prime implicant, it is necessary to treat x_1 as an unconditional 1. A similar procedure is carried out for all four variables. The pattern of 1's and 0's shown in Figure 16 can be treated as a permanent pattern or if desired as a programmable pattern which is set up when required. The function can be evaluated by successively driving the strap pairs assigned to x_1 through x_4. If a variable has the

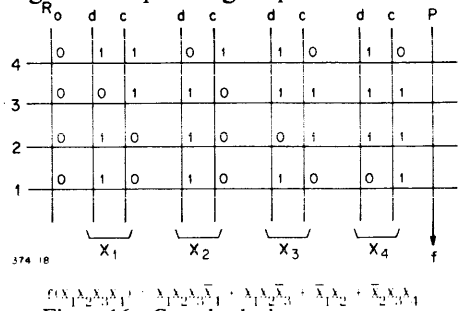


Figure 16 – Complex logic array pattern

value 1 or 0, the d or c strap of its pair is driven, respectively. The prime implicants will be formed under strap P, which is initially set to 1. On each of 4 successive cycles a transfer from the appropriate x-strap in combination with the R_o strap is executed. At the end of 4 cycles if strap P contains at least one 1, f is 1, otherwise it is 0.

The cellular technique described can clearly handle any combinational technique and with straightforward extensions can handle sequential functions.

SUMMARY

A promising magnetic logic-in-memory technique called bit steering has been described. The closed flux path and relatively thick film plated wire permit two source films to develop sufficiently large sense voltage to steer another film in a low impedance loop. Laboratory experiments have indicated basic technical feasibility. The inherent resistance and inductance of continuously plated wire optimized for conventional NDRO memories with a wire-write current of 25-35 milliamperes permit transfers over loops several inches in length; plated wire of this type is particularly suited for multiloop configurations. Low steering current wire (10 milliamperes) is required for substantially longer loops.

It is not possible at this time to give firm estimates on near future design details and array yield. Steady progress in the development of uniform films on wire and in plane fabrication give encouragement that the tolerances required on the delta currents due to imperfect cancellation at the transmitting elements and on the other described aspects of design can be attained.

Bit steering is inherently suited to parallel rather than general logic processing. The discussion in the paper has accordingly been limited to large scale memory-in-logic. It is expected that the complex interconnection patterns usually required in general purpose logic preclude the use of bit steering and such logic will be dominated by semiconductor technology.

ACKNOWLEDGMENT

The authors wish to express their appreciation to R. Tucker, R. McMahon and other members of the Research Operations at Blue Bell for fabricating experimental set-ups, to A. Turczyn for helpful discussions, and to W. M. Overn, W. W. Davis, R. A. White and others of Univac, St. Paul, for their formative work in bit steering. The authors are grateful for the interest shown in this work by Messers D. Haratz, A. Campi and B. Gray of the U. S. Army Electronics Command, Fort Monmouth, New Jersey.

REFERENCES

1 W B JOHNSON
Steering thin magnetic films
Proceedings Intermag Con 6 2 2 1963

2 M W GREEN
High speed components for digital computers
Final Engineering Report Part A SRI Project 2548 Prepared for Wright Air Development Division WPAFB Contract No AF 33 616 5804 1959

3 J G EDWARDS
Magnetic film logic
Proceedings IEE 112 40 1965

4 W E PROEBSTER H J OGUEY
High speed magnetic film logic
Digest of Technical Papers International Solid States Circuits Conference 23 1960

5 G W DICK D W DOUGHTY
A full binary adder using cylindrical thin film logic elements
Digest of Technical Papers International Solid State Circuits Conference 86 1963

6 G W DICK W D FARMER
Plated wire magnetic logic using resistive coupling
IEEE Trans on Magnetics MAG 2 343 1966

7 T R LONG
Electrodeposited memory elements for a nondestructive memory
J Appl Physics 31 1235 1960

8 J P McCALLISTER C F CHONG
A 500 nanosecond main computer memory utilizing plated wire elements
AFIPS Proceedings 29 FJCC 105 1966

9 T R FINCH S WAABEN
High speed DRO plated wire memory system
IEEE Trans on Magnetics MAG 2 529 1966

10 W J BARTIK C H CHONG A TURCZYN
A 100 megabit random access plated wire memory
Proceedings Intermag Conference 11 51 1965

11 I DANYLCHUK A J PERNESKI M W SAGAL
Plated wire magnetic film memories
Proceedings Intermag Conference 5 4 1 1964

12 Originally proposed by A CAMPI B GRAY of US Army Electronics Command Fort Monmouth New Jersey See Chow W F
Content addressable memory techniques
First Quarterly Progress Report January 31 1966 Prepared under Contract DA 28 043 AMC 01305 E by the Univac Division of the Sperry Rand Corporation Blue Bell Penn

A cryoelectronic distributed logic memory

by B. A. CRANE and R. R. LAANE
Bell Telephone Laboratories
Whippany, New Jersey

INTRODUCTION

The thin-film cryotron's ability to combine memory and logic functions in large, highly integrated circuit arrays has led to investigations of its use in implementing associative or content addressable memories.[1,2,3,4] This paper describes the design and operation of a cryoelectronic associative memory, a version of Distributed Logic Memory[5,6,7] consisting of 72, 8-bit associatively accessed cells and operating at around 0.4 million instructions per second. The memory array contains approximately 3400 cryotrons on four substrates. The substrates were fabricated by the General Electric Computer Laboratory, Sunnyvale, California, using their recently described hybrid technology;[8,9] and circuit design, array assembly, testing and operation was done by Bell Telephone Laboratories.

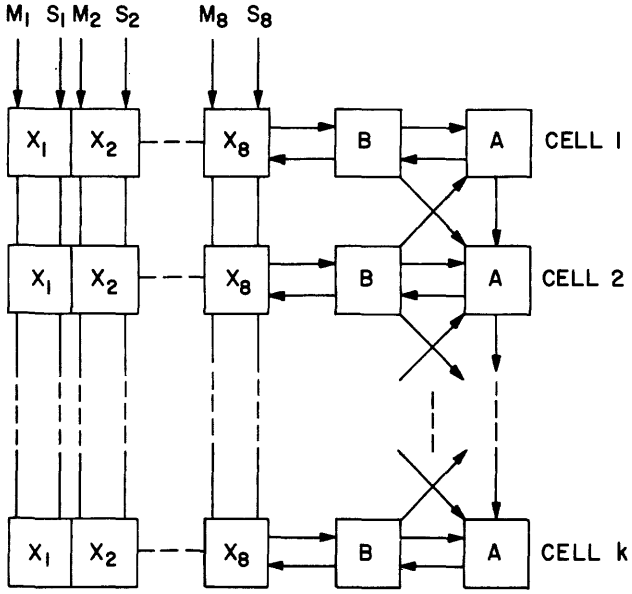


Figure 1 – Linear DLM array

A novel scheme for fault compensation was designed into the circuits so as to allow the use of substrates having faulty circuits or devices. Although the number of faults per substrate was typically low, this scheme proved indispensable in the assembly and successful operation of the final 72 cell array.

Distributed logic memory

Although detailed descriptions of DLM appear elsewhere,[5,6,7] a brief description here of the implemented version's organization will aid in following discussion. The implemented DLM is a linear array of cells such as shown in Figure 1. Each cell contains eight data flip-flops, X_1, X_2, ..., X_8; two control flip-flops, A and B; and combinatorial logic used in operating on cell contents.

The linear DLM array has all properties of a conventionally organized associative memory in that cell contents are operated on by MATCH, STORE, and READ commands. The MATCH command interrogates all cells simultaneously to find those containing the bit pattern specified on the match lines, M_1, M_2, ..., M_8. Each cell matching the input pattern is then marked as "active" by setting its A flip-flop to the "1" state. The STORE command writes a specified bit pattern into the cells via store lines S_1, S_2, ..., S_8. Two options are possible: either all cells or all active cells are written into simultaneously. In both the MATCH and STORE commands any bit position or combination of bit positions can be specified as "don't care." The READ command operates only on active cells, and when more than one cell is active, an algorithm must be used to select only one active cell at a time. In the implemented version, the contents of a selected active cell are then read out serial-by-bit.

The linear DLM array has the additional property that each cell communicates with its two nearest neighbors via its A and B flip-flops. This communication occurs in directional MATCH commands in which either the left or right neighbor of a matching cell is activated rather than the matching cell itself. Another, longer range form of communication

is afforded by the PROPAGATE command which extends activity, beginning at an already active cell, continuing through left neighboring cells whose contents match the input pattern, and ending at the first mismatching cell. Another added property of the linear DLM array is the ability to specify cell activity in the input pattern of the MATCH operation.

These additional properties enable storing information in the form of strings of characters on a character per cell basis. Lee and Paull[5] have shown how storing one binary coded alphanumeric character per cell gives an efficient means of storing and retrieving variable-length information strings, while Crane and Githens[7] have shown how storing binary coded numbers side-by-side in groups of cells on a bit per cell basis (each number being a string of binary characters) allows efficient arithmetic operations using MATCH and STORE commands.

These same properties, however, place a more stringent requirement on correct circuit operation since cells are not independent of one another as in a more conventional associative memory. To paraphrase: a string is no stronger than its weakest cell. However, as shown later, the array's natural redundancy (at the cell level) allows faulty cells to be deleted without changing the array's function (only its capacity is changed). Of course, when a cell is deleted, communication between the A and B flip-flops of the new neighbors must be established, and care must be taken to assure that drive line continuity is preserved. With proper layout and design, however, these tasks are easily accomplished.

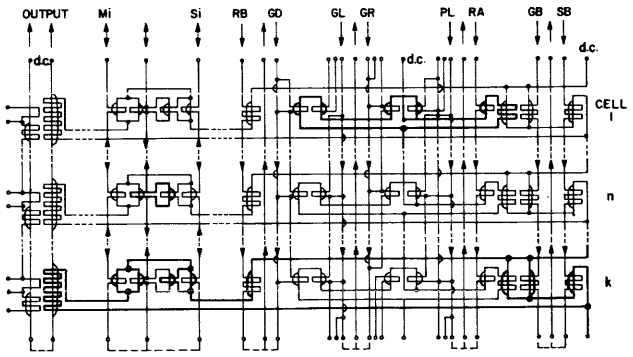


Figure 2 – Schematic drawing of linear DLM cells

Cell design

Figure 2 is a schematic drawing of three DLM cells, one interior cell and two end cells. The notation represents the cryotron gate as a semicircle and the cryotron control as a line crossing the gate. Note that most cryotrons have multiple control crossings. Only one of the eight identical data flip-flops is shown in each cell; and for clarity of description, one each of flip-flops A, B, and the data flip-flops are emphasized by heavier lines.

The B flip-flop, shown emphasized in cell k, is a loop stretching through the entire cell. Current supplied to all B flip-flops in series flows through either of the two legs of each loop and can be switched from one leg to the other by driving cryotron gates resistive while leaving others superconductive. For example, pulsing signal line RB (reset B) will drive current from the left hand leg (the "1" side) into the right hand leg (the "0" side) where it will remain after termination of RB (due to conservation of magnetic flux through a superconductive loop). Applying SB (set B) will then drive the current back to the "1" leg. As will be seen later, B must be in the "1" state when performing STORE and MATCH operations, and the result of a mismatch is to drive the current back to the "0" leg. Current in the "1" leg also drives the output cryotron resistive, thus registering an output.

A data flip-flop, shown emphasized in cell n, is a similar two-legged loop in which a persistent current is stored that circulates in the loop in either of two directions: clockwise for a stored "1" and counterclockwise for a stored "0". A "1" is stored by having B in its "1" state and sending current in line S_i and taking it out through the return line. The right leg of the storage loop is driven resistive by the current in B and the store current is therefore diverted to the left leg of the storage loop, where it remains when RB is applied. Terminating the store current, now, will cause a persisting current to circulate in a clockwise direction in the storage loop. Its magnitude is one-half that of the store current since the inductances of both legs of the storage loop are equal. Storing a "0" is done in the same manner except the direction of the store current is reversed.

To MATCH for a "0", B is set to "1" and a match current is applied to line M_i and removed through the return line. Because both legs of the storage loop have equal inductance, the match current will divide equally between them. If a "1" is stored, the match current in the left leg will combine with the persistent current to drive the "1" side of the B flip-flop resistive, thereby resetting B to "0". If a "0" is stored, the two currents cancel in the left leg of the storage loop and B remains set to "1". Matching for "1" is done by reversing the match current, and a don't care is accomplished by using no match current.

The A flip-flop, shown emphasized in cell 1, is similar to the B flip-flop. It is reset to "0", the left leg, by RA (reset A), and can be set to "1" by GD (gate down), GL (gate left), or GR (gate right) in conjunction with B being in the "1" state. The PROPAGATE command, setting A to "1" in cells

having B in the "1" state and being left neighbors of active cells, is accomplished with PL (propagate left). Cell activity is matched for by applying GB (gate B) rather than SB (set B) in preparation for the MATCH operation.

Fabrication

The cryotron circuits were fabricated by the General Electric Computer Laboratories using their recently described hybrid process.[8,9] The circuits are formed from five film layers on a 3¼ by 4 inch glass substrate. Sn gates are deposited through a mask onto an insulated ground plane. SiO is then deposited through a mask to insulate the gates from a top layer of Pb at certain points. The layer of Pb is then deposited over the entire substrate, contacting those parts of the Sn gates not covered by SiO. The Pb layer is then etched away (using photoresist techniques) to form the desired connections and controls. A thin layer of photoresist is then put over the entire substrate to act as a protective coating. The Sn gates are 18 mils wide and act either as the gate of an active cryotron (crossed by 1.6 mil wide Pb control lines) or as the underpass of a passive, crossover cryotron (crossed by a Pb connection line at least 10 mils wide).

The masks were made at Towne Laboratories, Somerville, N.J., from artwork made at BTL in accordance with GE's layout rules so as to assure compatibility with the fabrication process. Individual circuit parts (e.g., the Pb layer pattern of the data flip-flop) were cut on Mylar at 25 to 1 magnification and then reduced photographically. The final masters were then formed by step-and-repeat procedures. Figure 3 shown the top Pb layer pattern formed in this manner (the Pb remains in the white areas and is removed in the black).

Figure 4 is a photograph showing a finished DLM substrate with attached interconnection Pb strip line. Each substrate contains 20, 8-bit DLM cells comprised of 2100 active crossings (to form 940

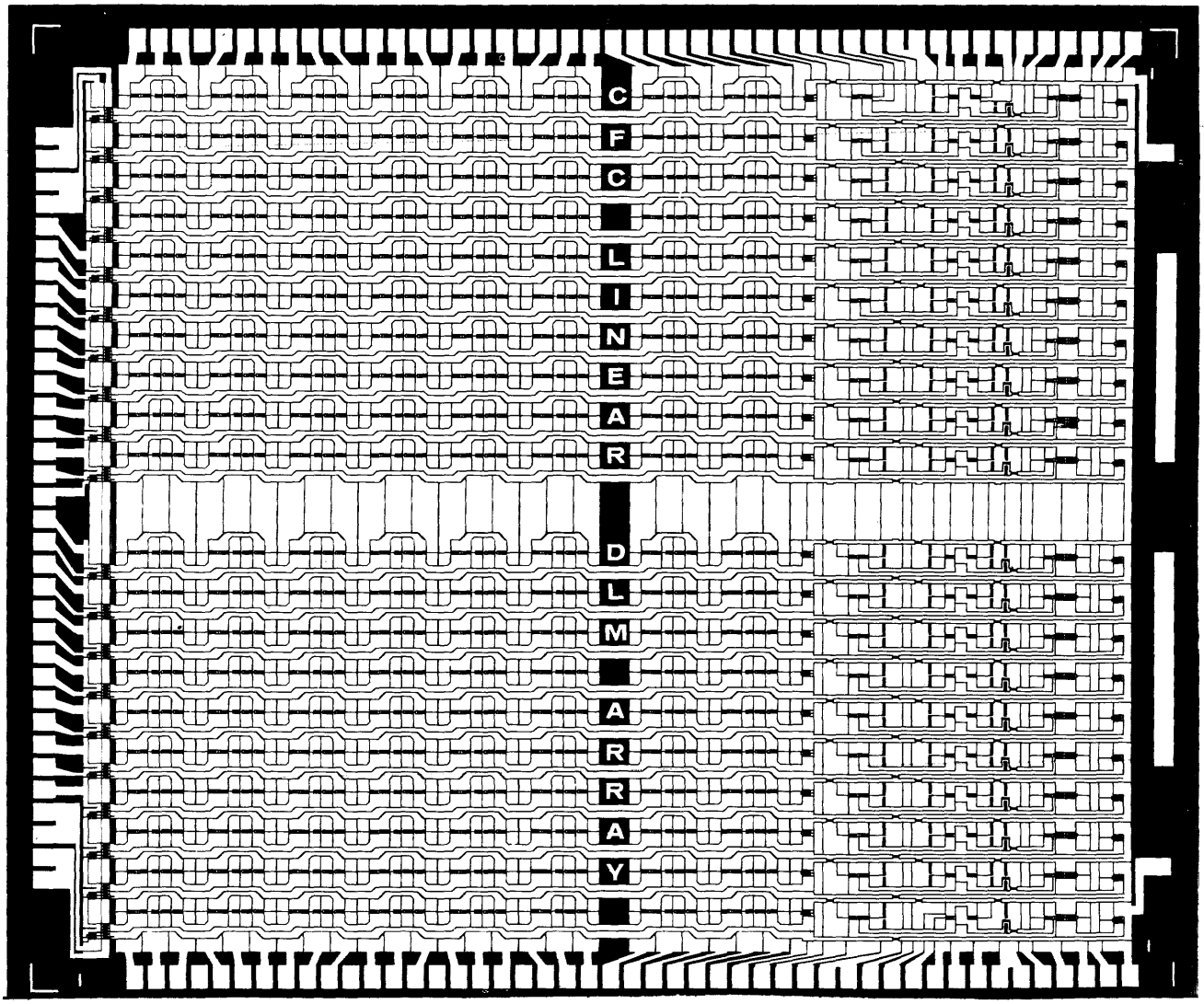


Figure 3 – Photo mask for top lead film layer

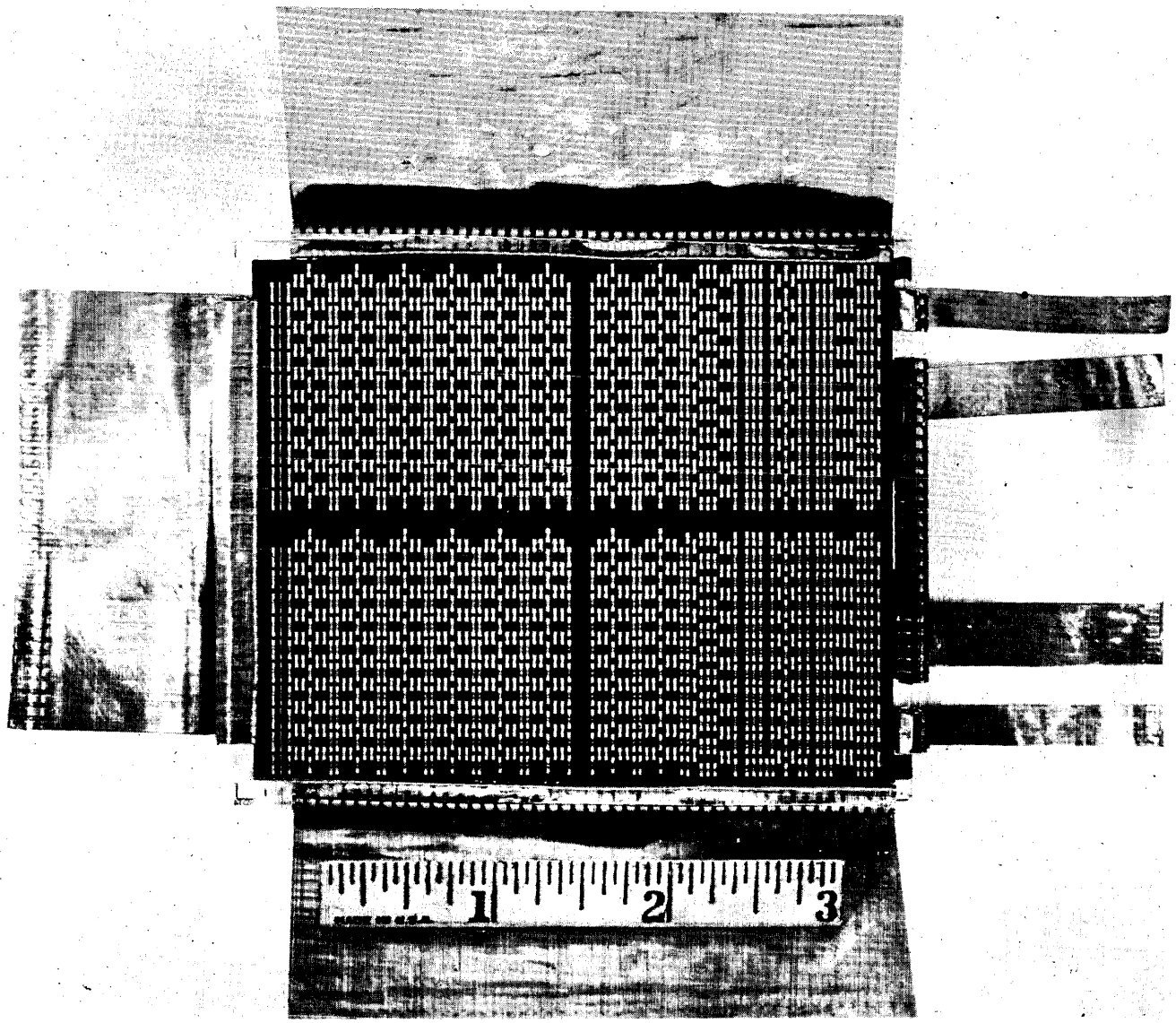


Figure 4—DLM substrate with attached interconnecting strip lines

cryotrons) and 2060 passive crossings (forming crossovers). The substrate also contains 80 active and 40 passive crossings used for device characterization measurements. The Pb strip line (Pb strips, 1 mil thick Mylar insulation, Pb ground plane) is clamped to the substrate by beryllium copper clamps. Each substrate has 92 connections for the DLM circuits and another 30 for the test devices.

Fault compensation

With cryotron circuits of this size and complexity, the probability of making a perfect substrate is small (but not zero). Consider, for example, the total length of the cryotron controls. With 2100 controls, each about 26 mils long, there is a total of about 4.6 feet of 1.6 mil wide Pb lines on each substrate, and a few opens are bound to occur. Likewise, the area of superimposed metal layers is roughly that of the substrate, making shorts quite probable. Therefore, in order to obtain reasonable yields, it is necessary to design the circuitry so as to be able to compensate for the faults that occur.

The predominant kind of fault in the DLM circuits was found to be open control lines. Device characteristics were sufficiently uniform to allow ±25% margin in operating currents, and shorts that occurred could almost always be eliminated by capacitive discharge techniques (this would sometimes create an open control, however). Because repairing a 1.6 mil wide control so that it will work again is very difficult, the tactic adopted was to disable the cell containing that control and to extend the neighboring cells' communication circuits over it. Open

controls occurring in the drive lines must, of course, be jumpered to preserve continuity.

The gate left, gate right, and propagate left circuits afford communication between cells, and, therefore, are the circuits that must be modified whenever cells are deleted from the array. To facilitate this modification, these circuits are realized as ladders having legs of unequal inductance. In the gate left circuit, for example (see Figure 5), the right verticals and the rungs have low inductance and the left verticals have high inductance (due to the controls). Current into the ladder will split according to inductance, the majority flowing in the right vertical whenever that branch is not resistive. In a cell having the B flip-flop in the "1" state, however, the corresponding right vertical is resistive, forcing the current to the left vertical (through the two rungs) and driving a gate in the "0" leg of the A flip-flop of the left neighboring cell resistive (activating the neighbor).

Suppose that cell 2 in Figure 5 is faulty. Then cell 1 can be made to communicate directly with cell 3 by first opening the rung between points W and X, and second assuring that the gate between points X and Y will never be driven resistive. The latter can be accomplished by causing an opening somewhere in the "1" side of cell 2's B flip-flop. A third, optional, compensation is to place a jumper between points X and W. This improves the integrity of the newly created left vertical (controls being narrow are susceptible to opens) but upsets the inductance realtionship. In practice, however, the control inductance is high enough to allow two consecutive controls to be jumpered without causing a harmful change in the inductance ratio. Jumpering, therefore, is in general advisable. The propagate left circuit is a compound ladder having three verticals per section but operates and is extended over faulty cells in a similar manner. Discontinuities in drive lines due to opens appearing in a control are repaired by placing a jumper between X and Y as shown in Figure 6.

Compensation for the deletion of a defective cell from the array using this technique requires the formation of 6 opens, and for an 8-bit cell having all drive lines jumpered for purpose of integrity, 31 jumpers. Figure 7 shows how the jumpers are formed; the end tabs of the controls where the Pb film contacts the Sn films underneath are 26 × 31 mils and afford ample area for jumper placement. Two techniques for making these jumpers proved successful. One, done at GE, consisted of removing the photoresist overlayer and depositing all jumpers simultaneously through a special mask. The other, done "in the field" at BTL, consisted of scraping away the photoresist at points to be jumpered and melting a small piece of Cerroseal solder between them. It was found that the solder would wet the photoresist, giving adequate mechanical support, and it remained superconducting during circuit operation. Figure 8 shows how the circuits are laid

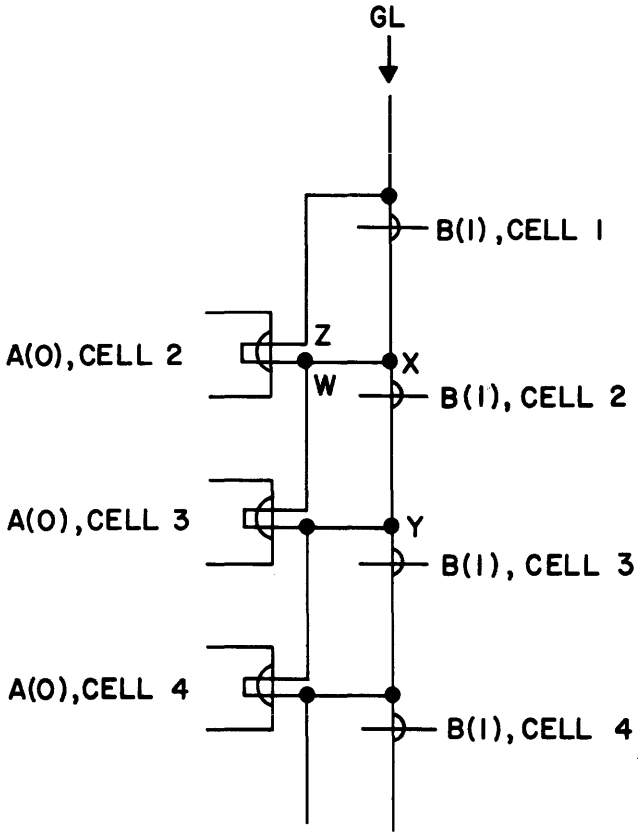


Figure 5 – Gate left circuit

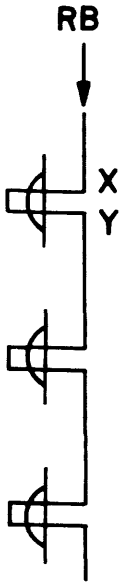


Figure 6 – Drive lines

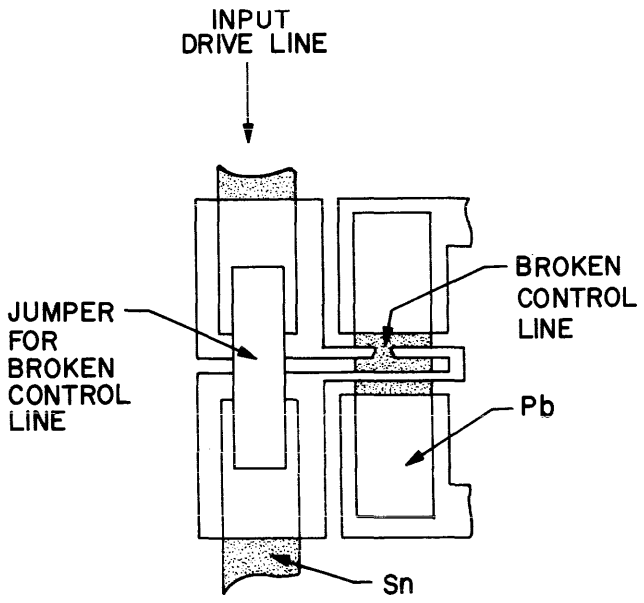


Figure 7 – Jumpering of control lines to regain drive line continuity

out so as to facilitate the creation of opens. There are 6 such points per cell, and the opens are made by scratching the film while viewing the circuit through a 20X microscope.

The usefulness of this technique can be seen from a sample of seven, 20-cell substrates that were fault compensated and selected for assembly into a four substrate array. One substrate contained 19 working cells, three substrates had 18, two had 17, and one had 16.

Array operation

After a series of device characterization measurements, manually operated function tests, and electronically generated test programs, four substrates were selected for assembly to form a 71-cell DLM array (8 cells having been deleted by fault compensation). Figure 9 is a photograph showing the 72-cell array mounted in its holder. The array operates at 3.55°K using 300 ma. supply current. Twisted pairs are used to carry the input, control, and output signals. Figure 10 shows superimposed the "0" and "1" outputs from the B flip-flop of one cell (after amplification by 100).

Figure 11 is an output waveform generated while exercising the 72-cell array with a program designed to test all circuit functions. To perform this program the cells are uniquely labeled 0 through 71, each cell storing its assigned number in data flip-flops X_1 through X_7, and all X_8 data flip-flops are reset to "0". The output waveform is from the "1" side of all B flip-flops in series so that its magnitude is proportional to the number of B flip-flops in the "1" state at that particular point in the program (the proportionality appears non-linear in the scope trace due to amplifier non-linearities and variations in output gate resistance).

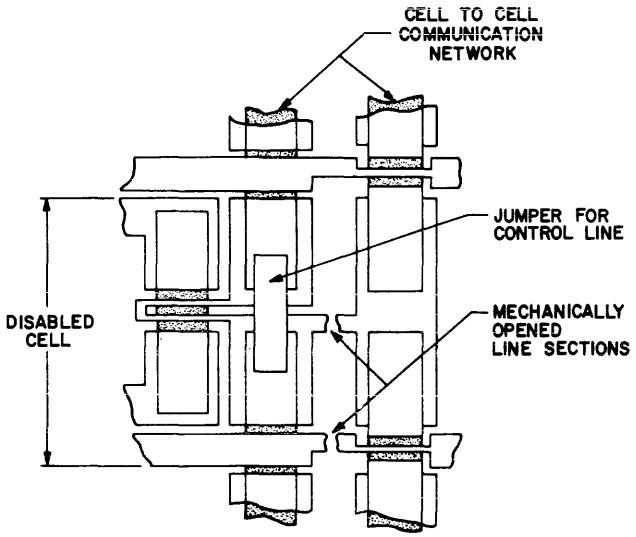


Figure 8 – Circuit layout for mechanically creating opened circuit line sections

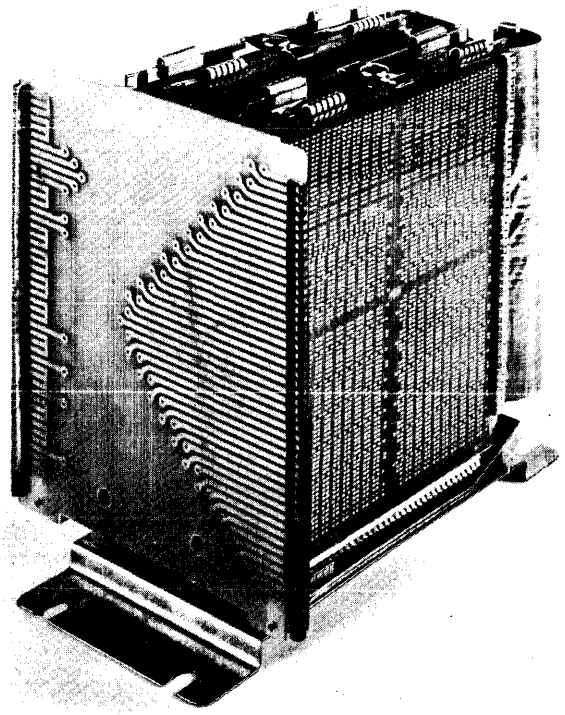


Figure 9 – 72 cell DLM array mounted in holder

A detailed description of the program is given in the appendix. Steps a through g check the match circuits at each bit location by matching for binary 35 (the 36th cell), one bit at a time, starting with the least significant bit position. Next, all A and B flip-flops are sequentially activated, first to the left of the matched cell location (by step h) and then to the right (by step i). This can be observed in Figure 11 as a

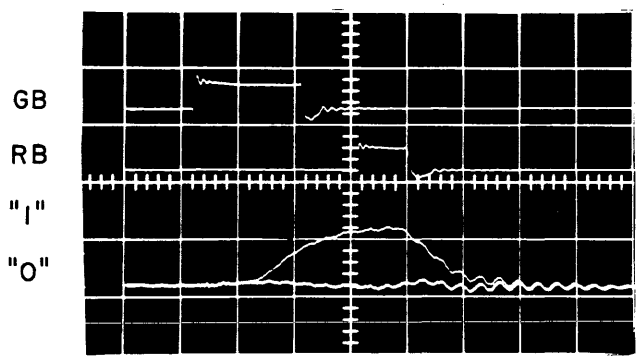


Figure 10 – Superimposed "0" and "1" outputs from the B flip-flop of one cell, after amplification by 100 (vert. 0.5 volts/cm.; horiz., 0.5μs/cm.)

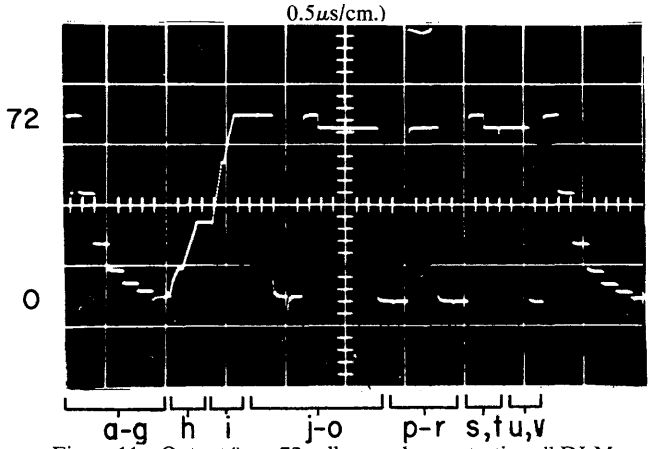


Figure 11 – Output from 72 cell array demonstrating all DLM circuit functions (horiz., 200μs/cm.)

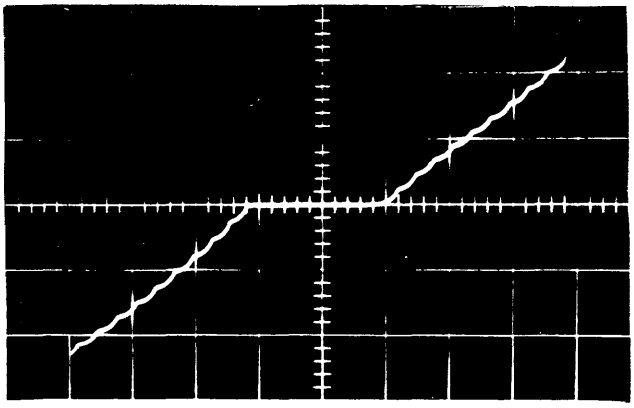


Figure 12 – Cell-to-cell communication (horiz., 5μs/cm.)

voltage ramp increasing incrementally as new B flip-flips are activated. An added time delay occurs when activity is transferred between substrates (because of added circuit inductance due to interconnecting strip lines). These delays can be observed as discontinuities in the voltage ramp and are shown in greater detail in Figure 12. Approximately 12μs are required for information transfer between substrates compared to 1.3μs on the substrate.

The propagate left circuit is tested (steps j through o) by first matching and activating cell 1, and then starting with the active cell, propagating activity toward the left as long as the cell to the left contains $X_7 = 0$. In steps p, q and r, $X_8 = 1$ is stored into all cells activated by propagation, and then a match is performed (in steps s and t) to insure correct storing. Finally, $X_8 = 0$ is stored back into matched cells (steps u and v), thereby completing the test cycle.

This test program was run at a relatively slow rate so as to allow time for rippling activity through 72 cells. As an example of maximum execution rate, a program composed equally of MATCH (3.2μs) and STORE (1.8μs) instructions was run at around 0.4 million instructions per second.

During a six-week period of testing and operation, this 72-cell array was cycled between room and liquid helium temperatures ten times. Although some of the original shorts would recur after a cycle, they could by removed by capacitive discharge and the array would show no degradation in performance.

CONCLUSION

This paper has described the design and operation of a 576-bit, cryoelectronic associative memory organized in the form of Distributed Logic Memory. Perhaps the most significant contribution of this work has been the demonstration of techniques for dealing with highly integrated circuit arrays containing faults. While these fault compensation techniques apply to a particular realization of a particular organization, it is suggested that the feasibility of applying some kind of fault compensation should be considered in evaluating organizations and technologies for implementation by large-scale integration techniques.

ACKNOWLEDGMENTS

We acknowledge with great pleasure the work of Mr. C. E. Doyle and Mr. B. Wentworth in designing, constructing, and operating the apparatus used for testing and operating the DLM array. We are also indebted to the members of the General Electric Computer Laboratory for their cooperation and skill in fabricating the cryotron substrates. Finally, we are grateful to Mr. J. A. Githens for his comments and encouragement during the course of this work.

APPENDIX

	Signal	*Result*
(a)	SB,RA	B = 1, A = 0 in all cells
(b)	$M_1 = 1$	B = 0 in cells where $X_1 = 0$
(c)	$M_2 = 1$	B = 0 in cells where $X_2 = 0$
(d)	$M_3 = 0$	B = 0 in cells where $X_3 = 1$

(e)	$M_4 = 0$	B = 0 in cells where $X_4 = 1$
(f)	$M_5 = 0$	B = 0 in cells where $X_5 = 1$
(g)	$M_6 = 1$	B = 0 in cells where $X_6 = 0$
(h)	GL, GB	A = 1, B = 1 in cells 37 through 72
(i)	GR, GB	A = 1, B = 1 in cells 35 through 1
(j)	SB, RA	B = 1, A = 0 in all cells
(k)	$M_1, M_2, \ldots, M_8 = 0$	B = 0 in all cells except cell 1
(l)	GB	A = 1 in cell 1
(m)	SB	B = 1 in all cells
(n)	$M_7 = 0$	B = 0 in cells where $X_7 = 1$
(o)	PL	A = 1 in cells 1 through 64
(p)	RB	B = 0 in all cells
(q)	GB, $S_8 = 1$	
(r)	RB, $S_8 = 1$	$X_8 = 1$ in cells 1 through 64
(s)	SB, RA	B = 1, A = 0 in all cells
(t)	$M_8 = 1$	B = 0 in all cells where $X_8 = 0$
(u)	GD, GB, $S_8 = 0$	
(v)	RB, $S_8 = 0$	$X_8 = 0$ in cells 1 through 64

REFERENCES

1 A E SLADE H O McMAHON
A cryotron catalog memory system
Proc EJCC 120 1956

2 V L NEWHOUSE R E FRUIN
Data addressed memory using thin-film cryotrons
Electronics 35-18 31 1962

3 J D BARNARD F A BEHNKE A B LINDQUIST R R SEEBER
Structure of a cryogenic associative processor
Proc IEEE 52 1182 1964

4 J P PRITCHARD JR
Superconducting thin-film technology and applications
IEEE Spectrum 3 46 1966

5 C Y LEE M C PAULL
A content addressable distributed logic memory with applications to information retrieval
Proc IEEE 51 924 1963

6 R S GAINES C Y LEE
An improved cell memory
IEEE Trans on Electronic Computers 14 72 1965

7 B A CRANE J A GITHENS
Bulk processing in distributed logic memory
IEEE Trans on Electronic Computers 14 186 1965

8 R E FRUIN A K OKA J W BREMER
A hybrid cryotron technology part I - circuits and devices
INTERMAG Abstracts 8.3 1966

9 C N ADAMS J W BREMER M L UMMEL
A hybrid cryotron technology part II - fabrication
INTERMAG Abstracts 8.4 1966

TRACE Time-shared routines for analysis, classification and evaluation*

by GERALD H. SHURE, ROBERT J. MEEKER
and WILLIAM H. MOORE, JR.

System Development Corporation
Santa Monica, California

*Research reported herein was conducted under SDC's independent research program and Contract No. DA 49-083 OSA-3124, Advanced Research Projects Agency, Department of Defense.

INTRODUCTION

Investigators often find it desirable to collect more data than are required to test preformulated hypotheses. Indeed, the experimenter with an empirical bent is reluctant to forgo the opportunity of collecting any data that might amplify or clarify his understanding. Particularly when anticipated relationships among variables fail to materialize (or, stated less elegantly, when predictions are not confirmed), he will wish to check various possibilities among supplementary data that may account for his negative results.

The on-line use of the computer to run an experiment now provides the experimenter with the ability to include more details of experimental processes and to record finer gradations of response than hitherto possible. As a result, the computer seems to be both a curse and a blessing to the researcher: The value of the increased amount of information that can be generated and recorded is often outweighed by the burden of organizing, collating, and assessing it. Moreover, much of the trial-to-trial data from these experiments are highly redundant, or irrelevant. If these data are also hierarchical, sequence-ordered, and of variable length—and such experimental data often have these characteristics—the problems of data management may become overwhelming.

The brute-force attack, even with available statistical computer programs, is rarely a satisfactory way of exploring the abundant, computer-recorded data. In a typical experiment of our own, we have had as many as 1,000 items of information for each of hundreds of subjects. If we did nothing more than calculate intercorrelations for each of these items of data, without considering combined indices, we would generate approximately one-half million correlation coefficients or cross-tabulations. The resulting stacks of computer printouts for such analyses are not infrequently measured in lineal inches. Techniques such as factor analysis are not particularly effective for inducing or discovering a fundamental order in the data, especially where reassessments frequently call for dividing the data in different ways, for omitting various subclasses of data from comparisons, for using different observational data or values in operational definitions, and, generally, for a great deal of data manipulation (recombining, regrouping, and recalculating) before new assessments can be made.

Consider the task, then, of classifying, grouping, and summarizing these data so as to identify summary and configurational indices which attempt to satisfy the criteria of reliability, specificity, validity, and relevance. Since an interplay of intuition and trial and error is frequently required to identify and develop such indices, they can become exceedingly costly in time and effort; and the task of iteratively sharpening and improving variable definitions can become impossibly burdensome. Often, under such conditions, the experimenter must either spend an inordinate amount of time to produce a thorough

analysis, or settle for the first few indices he develops (usually without attempting to assess the utility of these relative to a host of other possible indices). Yet he is aware that the insights for improving operational definitions of variables frequently are not forthcoming until one is well into the course of the analysis itself.

The TRACE program[1] was developed as we attempted to solve these kinds of data analysis difficulties encountered in the context of our computer studies of bargaining and negotiation. For the most part, these manipulations of the data precede the use of standard statistical analyses. TRACE is intended to assist the investigator in exploring relationships that may obtain among complex sets of data, from a number of different and newly suggested points of view, until he is satisfied that he has derived the optimal, or a satisfactory, operational definition. It also permits rapid checking of hypotheses about patterns and relationships for particular subsets of the data.

Description

To see how TRACE works, let's look for a moment at an abbreviated record of data from one of our own computer administered experiments in which, say, five hundred students play a game consisting of a sequence of five turns over a variable number of trials (see Figure 1). The computer keeps a complete,

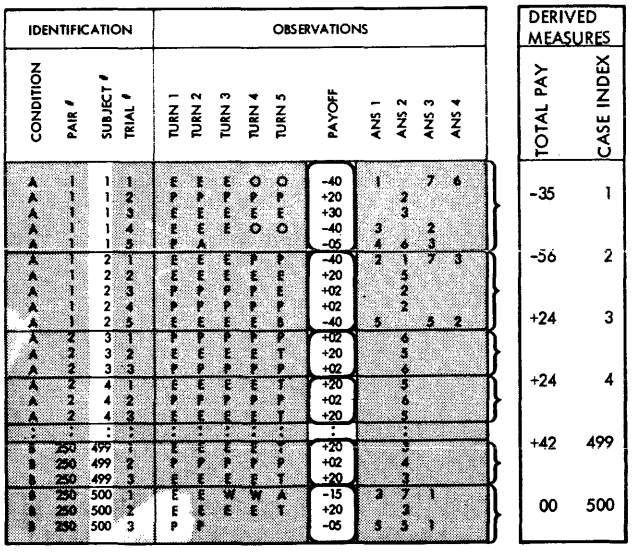


Figure 1 – Schematic of derived measure-sum of payoff values for each individual subject

detailed, trial-by-trial record for each subject that includes observations of behavior on successive turns in each trial, trial payoff, and trial-associated answers to questions. This record is loaded into TRACE which then performs like an automated data clerk who is prepared to calculate, count, classify, cross-tabulate, generate statistical indices, etc., and to keep records of these operations all according to the investigators' direction and specifications. The program uses the teletype to present the appropriate questions to the user. The user directs the program's operation by the answers he gives. A sample dialogue that corresponds to "Calculate each individual's total earnings" might proceed something like this:

DIALOGUE IN ENGLISH	*ACTUAL TRACE DIALOGUE*	*EXPLANATION*
TRACE: HOW DO YOU WANT THE DATA GROUPED FOR CALCULATION?	CASE INDEX	Every derivation requires a grouping of the data as a basis of calculation.
USER: A SCORE FOR EACH INDIVIDUAL SUBJECT IS DESIRED	*SUBJECT	The program then groups the data as bracketed in Figure 1.
TRACE: WHAT CALCULATION DO YOU WANT FORMED?	GIVE DERIVATION STATEMENT	The user can request any operation in the program's repertoire on any variable in the data record.
USER: THE SUM OF THE VALUES IN THE PAYOFF COLUMN	*IM = SUMPAYOFF //	
TRACE: WHAT DO YOU WANT TO NAME THIS NEW MEASURE?	IM001 IS PROGRAM NAME USER NAME (N)O CHANGE	The program assigns a name or permits the user to assign his own name.
USER: TOTAL PAY	*TOTAL PAY	

The derivation is now complete. Each individual's payoff is summed and the results stored along with the individual's identification number for future reference, as shown in Figure 1. The user can obtain the derived information whenever required by asking for TOTAL PAY. This, then, in a highly simplified way, illustrates the general pattern of interaction between the user and the program. The user selects an operation he wants performed; each selection triggers a series of questions designed to elicit the information the program needs to perform the operation. The user controls the program by selecting the operations, by choosing from the options presented,

and by supplying the necessary parameters. By allowing the investigator to modify the analyses as he goes along, TRACE permits an effective interplay between the investigator's conjectural and judgmental skills and the computer's capacity for rapid and accurate data processing.

TRACE operates as an on-line program in the Q-32 time-sharing system at the System Development Corporation.[2] It employs magnetic tapes and a 4,000K-disc file as auxiliary storage devices. Programs operating in the time-sharing system are limited to approximately 46K words and are swapped from drum to core according to the algorithm of the scheduler. The program design of TRACE is similar to the time-sharing system itself in that the TRACE executive, as one of the operational time-sharing programs, selects the subprograms from disc, reads them into an internal table, and transfers control to the subprogram.

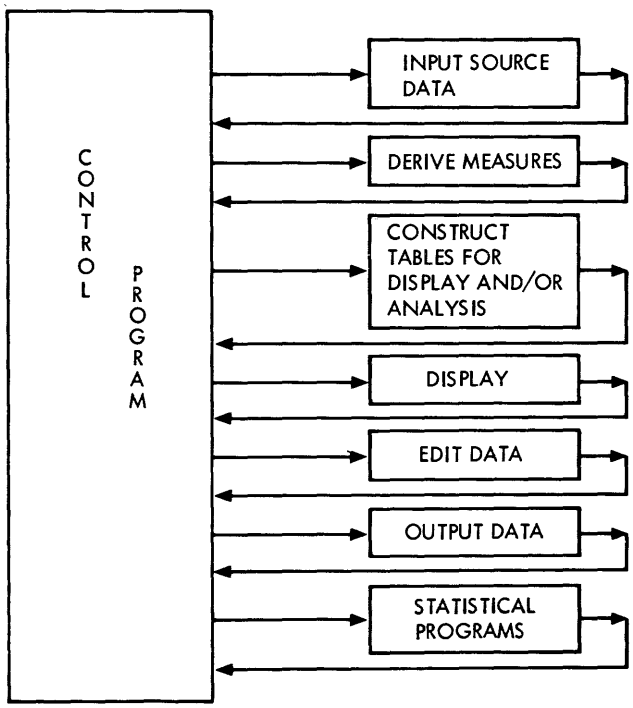


Figure 2 – Basic TRACE organization

Figure 2 depicts the basic program organization of TRACE. The user always operates in the context of the control program, through it he can select any one of the various subprograms for data processing. Each of the subprograms will then return the user to the control program for subsequent selection. Thus, the user exercises control by "bringing in" the desired subprogram.

The first step, of course, is to input the data. The data may be organized into as many as ten data sets. These may all be prestored on magnetic tapes generated from punched cards at the start of an analysis, or they may be added later. Small quantities of data can also be entered directly from the teletype. The added data sets can be more of the same, or "new" in the sense that they differ in content and format. If the separate data sets are related – if they pertain to the same individual, events, series of experiments, observations – they are, in effect, combined with the earlier data sets for analysis. The ability to combine data is made possible by providing each data set with one or more common variables to permit cross-referencing. If the data in one data set are not linked with the data in another in this fashion, the two will be treated independently of one another.

Every data base includes a dictionary containing a set of specifications for each variable and associating the variables in each set. The data include not only what is traditionally treated as data – the *observational* values based on measurements and observations – but also include *tag* values that indicate the sets with which each observational datum is associated (e.g., trial number, identification number for subject and pair, and experimental condition number in Figure 1). The observational and tag values in each data set are organized into units called data *blocks*. A data block is composed of one complete set of values, one value for each variable or subdivision of a variable specified in the dictionary. The identification values and observations contained in any row of Figure 1 constitute a data block.

For any particular derivation, the user is able to operate directly with the data organized into these block units, or to group the data into larger units along a dimension of the data record. To specify the grouping units, the user selects one of the variables (usually an identification variable) as the basis for aggregating the data. As noted in the illustration, this variable is designated as the CASE index. The values of the CASE index then define the functional grouping of the data blocks. Data blocks with CASE index values are treated as collective units. Thus, if the CASE index is "subjects," as in the example, all blocks of data with the same subject number as a value would be treated as a unit. If individual subjects were also members of other sets that might serve as primary dimensions (e.g., if an individual were also a member of a pair), then such analyses could be directly specified without requiring the data to be physically reconfigured. The CASE index is merely changed to "pair," and all blocks of data carrying the same pair identification number would be treated as a unit. This seemingly simple capability permits the user to select and modify the unit of analysis (the set of data points in the multivariate observation space) so as to satisfy a variety of approaches to the data.

Another step permits the user to restrict the data in an analysis to any specifiable subset, for example, to a specific set of trials or conditions. Following the specification of the data organization and subset, the derivation itself is made.

In the most elementary case, the derivation may be nothing more than a one-to-one transfer of source-variable values into a table for a derived variable. In the most complex derivation statement, 100 terms may be used in the definition of the derived variable. In general, the derivation procedure is designed to permit the user to define a new variable whose values will be based on logical and arithmetic combinations of values obtained from a set of existing variables.

The most general and complete form that the derivation statement can assume is the following:

$$A=B[IF\ C_1\ rel\ C_2\ con\ D_1\ rel\ D_2\ O\ TR\ E]$$

The expression A indicates the type of new variable to be derived alphanumeric, interger, or floating point may be specified.

Following this, the user indicates how he wants the measure computed. The = sign is shorthand for "set the measure equal to...." The deration formula may be a simple one and employ only expression B. The contents of B may also be limited to one term only—a variable name or even a single, specified value; in short, anything that would determine a value for the measure. On the other hand, the derivation may involve all the expressions B through E, and these expressions can be almost as complex as the user desires. Each of the expressions B, C_1, C_2, D_1, D_2, and E may consist of one or more variables, parameters, or constants. The user may preface any of these terms in the expression with functional operators including square root, log, trigonometric, statistical operators (sum, mean, median), and with counting operators that search multiple data entries in sequence to determine the frequency of occurrence of specified values.

The terms in each expression may also be related by the arithmetic connectives—addition, subtraction, multiplication, and division. An example of a derivation of a variable that involves only expression B might be as follows: Define a new variable, "degree of delinquency," and set it equal to the "frequency of arrest" divided by the "logarithm of the age of first arrest." In a more complex derivation, the user may wish to make the choice of computations depend on whether certain conditions are satisfied. If he wishes to do so, he will employ some or all of the parts of the general derivation expression that is bracketed—that is, all terms between and including IF and E. Two relational clauses, C_1 rel C_2 and D_1 rel D_2, joined by a connective, constitute the conditional portion of this specification. The relational operators that may be used as terms between the two C and the two D expressions are: "is equal to," "is not equal to," "is less than," "is less than or equal to," "is greater than," and "is greater than or equal to." The relational clauses may be joined by AND or OR connectives. Thus the general derivation statement reads: Set the value of the variable A equal to the expression B, IF C_1 stands in a certain relation (rel) to C_2, AND/OR D_1 stands in a certain relation (rel) to D_2, otherwise (OTR), set the value of the variable equal to the expression E. There is no restriction on the number of AND or OR clauses. A concrete example may help to illustrate a simple use of the conditional specification procedure. Let us assume that we wish to assign one of two social class designations based on information from four different measures. The derivation statement might look like the following expression:

```
AM = UPPER IF MEAN INCOME 1-20 GR
     65,000 AND RESIDENCE GQ 150,000
     OR OCC EQ EXECUTIVE AND NAME
     EQ FORD OTR BOURGEOIS
```

The statement says: Define a new alphanumeric measure (AM), social class, and set it equal to "upper" IF the mean income of the individual over the first twenty years of his work is greater than $65,000 and his residence is worth more than $150,000, or if his occupation is classified as executive and his name is Ford; otherwise, if neither of the conjuncts is satisfied, assign him to the bourgeois. This derivation will assign a class value for every unique case index.

The example can also be used to illustrate another powerful capability of the derivation procedure. A variable may contain a subdivided string of symbols. In our example, each of the 20 annual income figures is treated as a separate datum, and a mean is calculated from these values. The user may directly specify the use of any specified subset of data in a string variable. Furthermore, each subdivision may be treated as a separate value, as in the above example, or the string may be aggregated into larger units. For instance, the entire string may be treated as a single datum. This capability permits the derivation program to be used as a device for detecting sequence patterns in the data—by sliding a window across successive sets of entries and sensing for particular patterns.

The example used to illustrate the derivations of a social class index illustrates, perhaps too facetiously, the arbitrariness of many of our operational definitions in the social sciences. Would the index be better if we substitute other criteria for those used? Should

we add the names of Du Pont and Rockefeller to that of Ford? Is there a better set of constants for the expression? If a hunch or insight suggests a different set, it takes only a minute or two to derive a new variable with TRACE.

The construct and display subprograms permit the user to examine the distribution of the newly derived variables and to relate each to other criterion variables. The construct program offers a number of criteria for partitioning each variable into intervals or classes. The display program provides a means of evaluation—to see if there are anomolous distributions which will require redefinition or adjustment; or, in the bivariate distribution, to make preliminary assessments of a measure's descriptive or predictive potential. In other words, the display function is not primarily intended as the terminal step in analysis, but as a basis for further refinement and exploration. A number of statistical indices and simple procedures for identifying outliers are available along with a number of other features for aiding the user to augment and check his insights and further refine his measures as he re-examines his data.

One additional general comment should be made. Such a program greatly magnifies the possibility for chance to create ostensibly significant results, since the relationships that are found will clearly not be formulated independently of the data. The more statistically sophisticated data analysts may be alarmed by the prospect that TRACE users will greatly inflate their reportings of findings because of misleading probability levels. This is a danger, but, as the statistician John Tukey[3] has noted, this has not proved to be a problem in the physical sciences where the results of praying over the data have proved to be of greatest importance. When these reanalyses are suggestive, the results are thought of as something to be put to further test in another experiment, as *indications* rather than *conclusions*. Let me quote Tukey directly.

> *"We need to face up to the need for iterative procedures in data analysis. It is nice to plan to make but a single analysis, to avoid finding that the results of one analysis have led to a requirement for making a different one. It is also nice to be able to carry out an individual analysis in a single straight-forward step, to avoid interation and repeated computation. But it is not realistic to believe that good data analysis is consistent with either of these niceties. As we learn how to do better data analysis, computation will get more extensive, rather than simpler, and reanalysis will become much more nearly the custom."*

TRACE is a computer program which makes it easier to heed the implication of Dr. Tukey's advice.

REFERENCES

1 W H MOORE JR R J MEEKER G H SHURE
TRACE—Model I: timeshared routines for analysis classification and evaluation
SDC document TM-2621 1965

2 J I SCHWARTZ E G COFFMAN C WEISSMAN
A general-purpose time-sharing system
Proceedings of the spring joint computer conference
pp 397-411 1964

3 J W TUKEY
The future of data analysis
Ann Math Statist 33 1-67 1962

Degradation analysis of digitized signal transmission

by J. C. KIM and E. P. KAISER

Melpar, Inc.
Falls Church, Virginia

INTRODUCTION

In communication systems signals are subjected to various kinds of noises. Although extensive studies of error control methods have been conducted, there is little information available as to the effects of noise upon the recovered and processed signal.

This paper presents some results of the degradation analysis of digitized, sampled analog signals transmitted through a digital communication system[1] shown in Figure 1. This analysis was accomplished by a theoretical investigation and a digital computer simulation of the problem.

The analog signal was a Rayleigh function. Its sampled version, was digitized and mixed with noise consisting of an independent, uniformly distributed random binary sequence. At the receiver the corrupted digital signal was converted to its analog equivalent and analyzed.

Two different measures of degradation were used in the analysis. The first measure was the normalized rms difference between sample pairs; the second measure was the normalized rms difference between the spectral components associated with each pair. This second measure was selected in order that one, using a computer to perform a spectral analysis, could better relate the degradation to the channel noise.

ANALOG SIGNAL CONSIDERED FOR DEGRADATION ANALYSIS

The signal considered in this paper is a waveform of the fundamental Rayleigh[2] mode expressed in equation (1) and shown in Figure 2,

$$f(t) = \frac{t \cdot e^{-t^2/\alpha}}{\alpha} \qquad (1)$$

where α is the attenuation constant. It is a well-known fact that the fundamental Rayleigh mode has its energy concentrated in the range of 0.15 to 10 cps. Using Nyquist's[3] Sampling Theorem, the rate of sampling is chosen as 20 samples per second.

Degradation analysis in sampling

Consider the analog signal f(t) from equation (1). When the analog signal f(t) is sampled at the rate of n samples per second, the sampled data can be represented as:

$$f(t) \rightarrow X(iT) \qquad (2)$$

where $T = \frac{1}{n}$ seconds, and where $i = 0, 1, 2, \ldots, m$. The binary sequence can be obtained by expanding X(iT) into k bits. Then, the rate of the digitized signal $IX_i(I)$ is nk bits per second. These curves are shown in Figure 3(a) and (b).

Whenever a binary sequence $IX_i(I)$ is transmitted through a communication system such as that shown in Figure 1, it will be corrupted by noise. For uniformly distributed random binary noise, the communication system may be theoretically modeled after the binary symmetrical channel. The binary symmetrical channel (shown in Figure 4) is the model of the overall transmission link; the conditional probabilities of a 0 or a 1 having been sent, given that a 0 was received, are p and q, respectively.[4]

In this paper, the system noise $IN_i(I)$ is a random binary sequence, as shown in Figure 3(c). The received binary sequence $IY_i(I)$ with rate nk (bits/second) is the logical sum of $IX_i(I)$ and $IN_i(I)$ given in equation (3) and shown in figure 3(d).

$$IY_i(I) = IX_i(I) \oplus IN_i(I) \qquad (3)$$

The signal recovered by digital-to-analog conversion Y(iT) is

$$Y(iT) = \sum_{I=1}^{k} IY_i(I) \cdot 2^{(3-I)} \qquad (4)$$

as shown in Figure 3(e). Note that the binary point is between the third and fourth leftmost bits of $IY_i(I)$.

Truncation errors in sampling

An error in the representation of a sampled signal X(iT) is inevitable. The amount of error E is found by taking the root mean square of the difference of the original and the recovered signals.[5,6]

$$E = \sqrt{\sum_{i=1}^{m} [T \cdot Y(iT) - T \cdot X(iT)]^2} \quad (5)$$

The normalized error is obtained by taking the ratio of the rms value to the total area (a) under the original digitized curve X(iT).

$$a = \sum_{i=1}^{m} T \cdot X(iT) \quad (6)$$

The normalized error $\bar{E}$ is

$$\bar{E} = \frac{\sqrt{\sum_{i=1}^{m} [Y(iT) - X(iT)]^2}}{\sum_{i=1}^{m} X(iT)} \quad (7)$$

The maximum truncation error associated with each sampling point is

$$\Delta_{max} = 2^{-k'}$$

where k′ is the number of bits to the right of the binary point of the binary represented of the sampled signal X(iT). The maximum truncation error E_{trunc} is obtained by using equation (5)

$$E_{trunc} = T\sqrt{\sum_{i=1}^{m} \{[X(iT) + \Delta_{max}] - X(iT)\}^2}$$

$$= T\sqrt{m}\Delta_{max} \quad (8)$$

The total area (a) under the curve is obtained from equation (6)

$$a = \sum_{i=1}^{m} X(iT) \cdot T \quad (9)$$

and the normalized truncation error $\bar{E}_{trunc}$ is obtained by using equation (7).

$$E_{trunc} = \frac{\sqrt{m}\,\Delta_{max}}{\sum_{i=1}^{m} X(iT)} = \frac{\sqrt{m}\,2^{-k'}}{\sum_{i=1}^{m} X(iT)} \quad (10)$$

Noise error

The error due to transmission noise is found by taking the root mean square of the difference of the original and noise-corrupted signals. This error also includes the truncation error. The noise error and the normalized noise error are obtained by using equations (5), (6) and (7).

The maximum noise error bound can be found as follows: The maximum noise error associated with each transmitted digital sequence is

$$\delta_{max} = 2^{(k''+1)} \cdot q \quad (11)$$

where k″ is the number of bits to the left of the binary point of the transmitted sequences. The noise error E_{noise} is obtained from equation (5)

$$E_{noise} = T\sqrt{\sum_{i=1}^{m} \{[X(iT) + \Delta_{max} + \delta_{max}] - X(iT)\}^2}$$

$$= T\sqrt{m}\,(\Delta_{max} + \delta_{max}) \quad (12)$$

and the normalized noise error, E_{noise}, is obtained using equation (7)

$$E_{noise} = \frac{\sqrt{m}\,(\Delta_{max} + \delta_{max})}{\sum_{i=1}^{m} X(iT)} = \frac{\sqrt{m}\,[2^{-k'} + 2^{(k+1'')} \cdot q]}{\sum_{i=1}^{m} X(iT)} \quad (13)$$

Degradation analysis of Fourier transform

The Fourier coefficients for the original signal are defined[6] as

$$a_n = \frac{2}{m}\sum_{i=0}^{m-1} X(iT)\cos\frac{2\pi ni}{m};\ n = 0, \ldots, \frac{m}{2} - 1 \quad (14)$$

$$b_n = \frac{2}{m}\sum_{i=0}^{m-1} X(iT)\sin\frac{2\pi ni}{m};\ n = 0, 1, \ldots, \frac{m}{2} - 1 \quad (15)$$

The original time function f(t) is represented as

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{m-1}\left(a_n\cos\frac{2\pi ni}{m} + b_n\sin\frac{2\pi ni}{m}\right) \quad (16)$$

and the power spectrum is

$$\rho_n = \sqrt{a_n^2 + b_n^2};\ n = 0, \ldots, \frac{m}{2} - 1 \tag{17}$$

The Fourier coefficients for the recovered signal are

$$a_n' = \frac{2}{m} \sum_{i=0}^{m-1} Y(iT) \cos\frac{2\pi ni}{m};\ n = 0, \ldots, \frac{m}{2} - 1 \tag{18}$$

$$b_n' = \frac{2}{m} \sum_{i=0}^{m-1} Y(iT) \sin\frac{2\pi ni}{m};\ n = 0, \ldots, \frac{m}{2} - 1 \tag{19}$$

The recovered time function f' (t) is represented as

$$f'(t) = \frac{a_0'}{2} + \sum_{n=1}^{m-1} \left(a_n' \cos\frac{2\pi ni}{m} + b_n' \sin\frac{2\pi ni}{m} \right) \tag{20}$$

and the power spectrum is

$$p_n' = \sqrt{a_n'^2 + b_n'^2} \tag{21}$$

Power spectrum truncation error

As discussed in "Truncation Errors in Sampling," the spectrum error is found by taking the root mean square of the difference of the power spectra of the original and the recovered signals. The error of the cosine coefficients E_{a_n} is

$$E_{a_n} = \sqrt{(a_n - a_n')^2} = \sqrt{\left(\frac{2}{m}\right)^2 \left[\sum_{i=0}^{m-1} Y(iT) \cos\frac{2\pi ni}{m} \sum_{i=0}^{m-1} X(iT) \cos\frac{2\pi ni}{m} \right]^2}$$

$$\leqq \frac{2}{m} \sqrt{\sum_{i=0}^{m-1} \left\{ [Y(iT) - X(iT)] \cos\frac{2\pi ni}{m} \right\}^2} \tag{22}$$

using the Schwarz inequality.[7] The upper bound of E_{a_n} is

$$E_{a_{n\max}} = \frac{2}{m} \sqrt{\sum_{i=0}^{m-1} [Y(iT) - X(iT)]^2} \tag{23}$$

since $\max\left(\cos\frac{2\pi ni}{m}\right) = 1$. Similarly, the error in the sine coefficients E_{on} is

$$E_{b_n} \leqq \frac{2}{m} \sqrt{\sum_{i=0}^{m-1} \left\{ [Y(iT) - X(iT)] \sin\frac{2\pi ni} \right\}^2} \tag{24}$$

and the upper bound of E_{b_n} is

$$E_{b_{n\max}} = \frac{2}{m} \sqrt{\sum_{i=0}^{m-1} [Y(iT) - X(iT)]^2} \tag{25}$$

The upper bound of the *jth* error term in the power spectrum $E_{\rho n}$ is

$$E_{\rho_j} = \sqrt{E_{a_{j\max}}^2 + E_{b_{j\max}}^2} = \frac{2}{m} \sqrt{2 \sum_{i=0}^{m-1} [Y(iT) - X(iT)]^2} \tag{26}$$

The maximum total error of the power spectrum E_ρ is

$$E_\rho = \sum_{j=0}^{N} \Delta f\, E_{\rho_j} = \sum_{j=0}^{N} \frac{2\Delta f}{m} \sqrt{2 \sum_{i=0}^{m-1} [Y(iT) - X(iT)]^2} \tag{27}$$

where $N = m - 1$ and Δf is the incremental frequency $\binom{N}{M}$. The total area under the power spectrum curve a_ρ is

$$a_\rho = \sum_{j=0}^{N} \rho_j\, \Delta f \tag{28}$$

The normalized power spectrum error $\bar{E}_\rho$ is obtained by taking the ratio of the total error in the power spectrum E_ρ to the total area under the power spectrum curve a_ρ

$$\bar{E}_\rho = \frac{E_\rho}{a_\rho} = \frac{\frac{2^{3/2}}{m} \sum_{j=0}^{N} \sqrt{\sum_{i=0}^{m-1} [Y(iT) - X(iT)]^2}}{\sum_{j=0}^{N} \rho_j} \tag{29}$$

The power spectrum error due to truncation $E_{\rho_{trunc}}$ is obtained by using equations (7) and (27)

$$\bar{E}_{\rho_{trunc}} = \frac{2^{3/2}\,\Delta f}{m} \sum_{j=0}^{N} \sqrt{\sum_{i=0}^{m-1} \Delta_{max}^{\,2}} = \frac{2^{3/2}\,\Delta f\,(N+1)\,\Delta_{max}}{\sqrt{m}} = \frac{2^{3/2}\,\Delta f\,(N+1)\,2^{-k'}}{\sqrt{m}} \tag{30}$$

and the normalized power spectrum error due to truncation $\bar{E}_{\rho_{trunc}}$ is obtained by using equation (29)

$$E_{\rho_{trunc}} \frac{2^{3/2}\,\Delta f\,(N+1)\,2^{-k}}{\sqrt{m} \sum_{j=0}^{N} \rho_j} \tag{31}$$

Power spectrum noise error

The error due to transmission noise is found in the same way that error due to noise and error due to truncation were found. The power spectrum noise error is obtained by using equation (27)

$$E\,\rho_{noise} = \frac{2^{3/2}\,\Delta f}{m} \sum_{j=0}^{N} \sqrt{\sum_{i=0}^{m-1} (\Delta_{max} + \delta_{max})^2} = \frac{2^{3/2}\,\Delta f\,(N+1)\,(\Delta_{max} + \delta_{max})}{\sqrt{m}} = \frac{2^{3/2}\,\Delta f\,(N+1)\,[2^{-k'} + 2^{(k''+1)}\,q]}{\sqrt{m}} \tag{32}$$

and the normalized power spectrum error $\bar{E}_{\rho_{noise}}$ is obtained by equation (29)

$$E\,\rho_{noise} = \frac{2^{3/2}\,(N+1)\,[2^{-k'} + 2^{(k''+1)}\,q]}{\sqrt{m} \sum_{j=0}^{N} \rho_j} \tag{33}$$

COMPUTER SIMULATION OF DEGRADATION ANALYSIS

We now describe the results of a computer simulation which considers analog-to-digital conversion of the sampled signal, the corruption of the binary data by a random binary sequence, the digital-to-analog conversion (recovery) of the received binary signal, the computation of the sine and cosine coefficients of the recovered sampled signal, the computation of the power spectrum of the original and the recovered signals, and the computation of error. Included are the results of computer evaluations of the theoretical error bounds.

Figure 5 is a simplified flow graph of the simulation. Part 1 is concerned with storing on tape the transmitted and received sample values of the analog signal, while Part 2 is concerned with using the taped data and the determination of error.

Figure 6 contains a graph of the actual sampled analog signal. The analog signal was sampled at the rate of 20 samples/second, yielding 100 samples in all.

Figure 7 contains graph displays of X and Y for various values of q. For q=0, 10^{-5}, 10^{-4}, 10^{-3}, and 5×10^{-3}, there is little visible difference between the transmitted and the recovered sample values, X and Y. From Figure 9 (corresponding to $q=10^{-3}$), the rms error between X and Y is about 10^{-2}. For $q\geq10^{-2}$, the difference between X and Y becomes noticeable, as seen in Figure 7, sheets 4 and 5. From Figure 9 ($q\geq10^{-2}$), the rms error is greater than 9×10^{-2}.

As shown in Figure 8 ($q\geq10^{-3}$), the difference between the original and the reconstructed power spectra becomes appreciably noticeable; and, as shown in Figure 10, the error between the spectra is greater than 7×10^{-2}.

Note that in both Figures 9 and 10, the error in the simulated case at times exceeds the theoretical limit by a magnitude of order of about 1. This is most likely due to the possibility that the random binary noise generator is generating slightly more noise within the relatively short sample period. On the average, if more runs were made, it is expectd that the error would approach the theoretical limit. In any case, conclusions based upon the simulation would be conservative, since the error in the simulated case exceeds the theoretical error.

DISCUSSION AND CONCLUSIONS

We now present some results of degradation analysis of analog signals through a digital communication system.

Two measures of degradation were considered; i.e., (1) rms value of the difference between the transmitted sampled analog function and the received analog function, and (2) rms value of the difference between the spectra of the transmitted and received signals. Based upon the results just covered and assuming that an error of no more than 0.01 can be tolerated with either measure, an error rate of no more than 1 bit in 10,000 bits can be tolerated. Truncation error (the error due to limiting the digital equivalent of the sampled analog signal to 12 bits) is negligible.

In this analysis, the binary symmetric channel was used as the model of the digital communication system. The binary symmetric channel is a good model of a system which encounters independent uniformly distributed, random bit errors. For those systems whose bit errors are not independent, the burst noise or Gilbert channel should be considered. Through the use of proper encoding and error correction, it is possible to reduce effectively the burst noise channel to the binary symmetric channel. In the case of either channel, the above-stated error rate should be the goal of any error correction system.

ACKNOWLEDGMENTS

The authors would like to thank Dr. A. R. Schuler of Melpar, Inc., for his comments pertaining to this work.

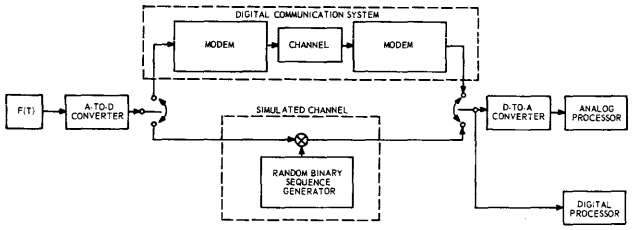


Figure 1—Theoretical model used in degradation analysis of a digital communication system

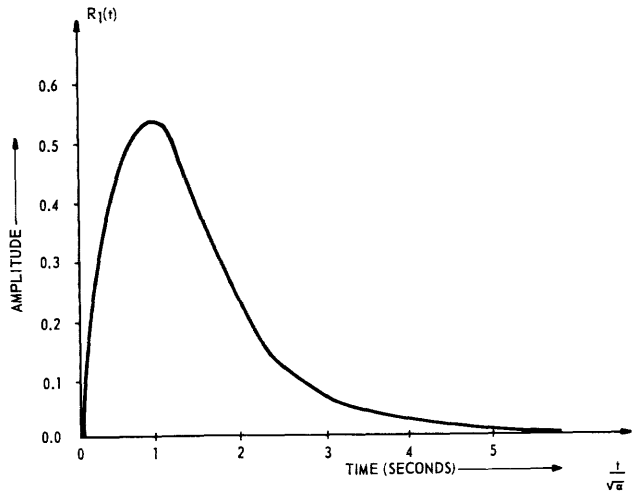


Figure 2—Function of fundamental Rayleigh mode

REFERENCES

1 J. C. KIM and E. P. KAISER
High-data-rate transmission through band-limited channels
Melpar Technical Note T68926.103 June 1966

2 D. MIDDLETON
Statistical communication theory
McGraw-Hill Book Company, Inc., New York p. 40 1 1960

3 H. NYQUIST
Certain topics in telegraph transmission theory
AIEE Trans. April 1928

4 R. FANO
Transmission of information
MIT Press p. 129 1961

5 A. PAPOULIS
Error analysis in sampling theory
Proc. IEEE July 1966

6 R. B. BLACKMAN and J. W. TUKEY
The measurement of power spectra
Dover Publication 1958

7 G. H. HARDY, J. E. LITTLEWOOD and G. PLOYA
Inequalities
Cambridge University Press, New York p. 3 1964

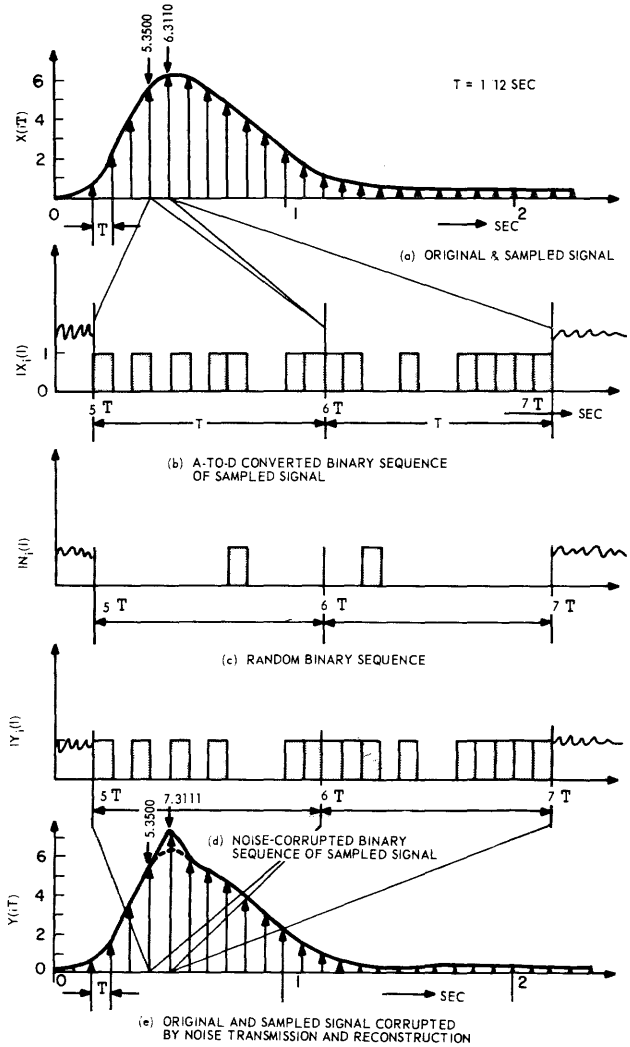


Figure 3—Typical transmission of an analog signal through a digital communication system

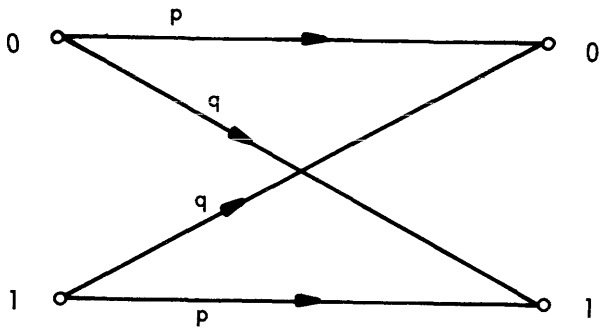


Figure 4—Transition diagram of the binary symmetrical channel

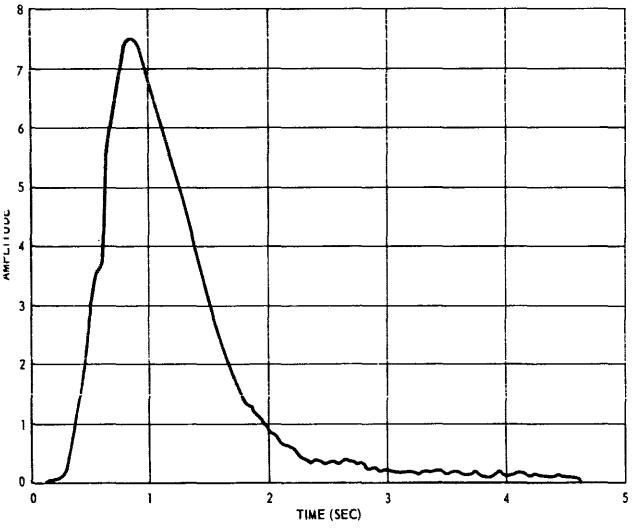


Figure 6—Analog time function of the Rayleigh fundamental mode

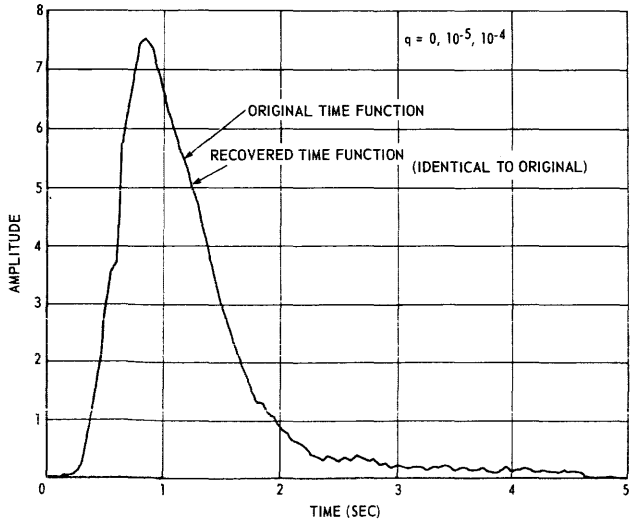


Figure 7—Original and noise-corrupted analog time functions of theRayleigh fundamental mode

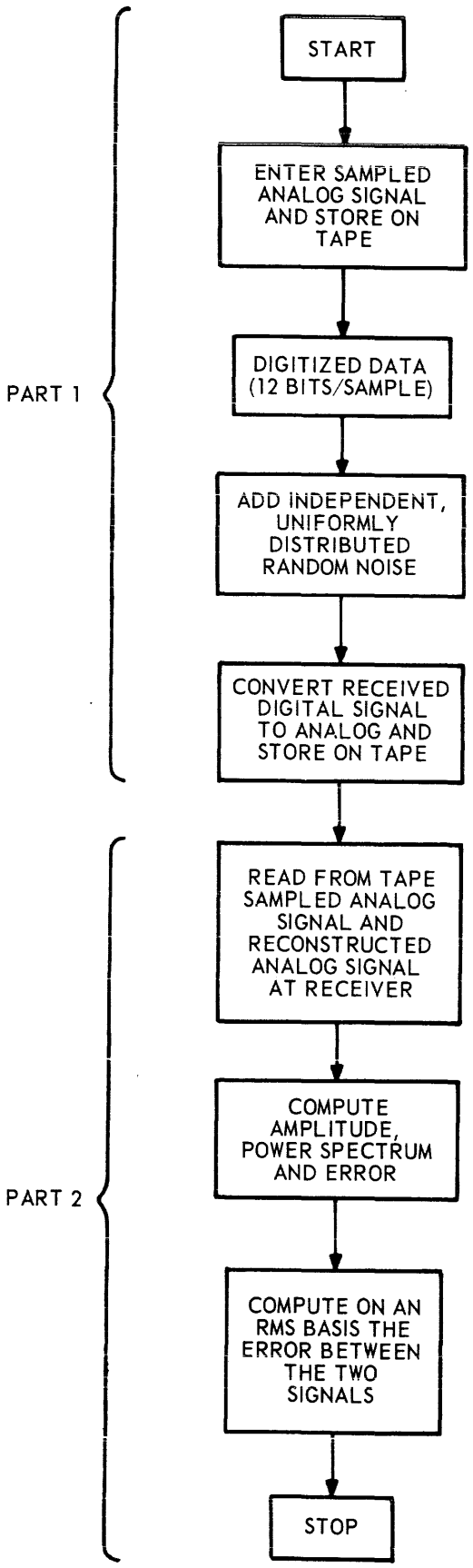


Figure 5—Simplified flow graph of computer simulation of a digital communication system

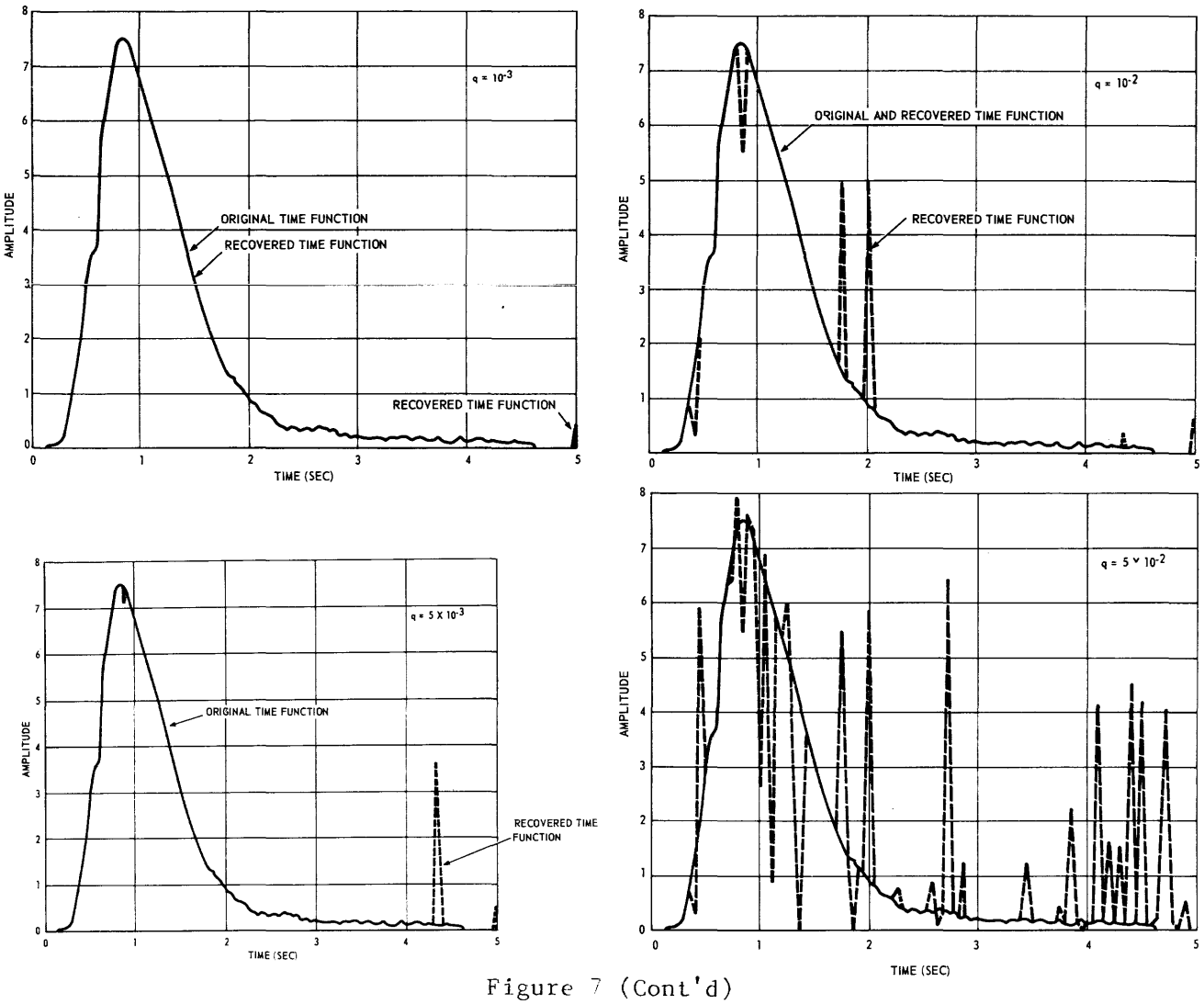


Figure 7 (Cont'd)

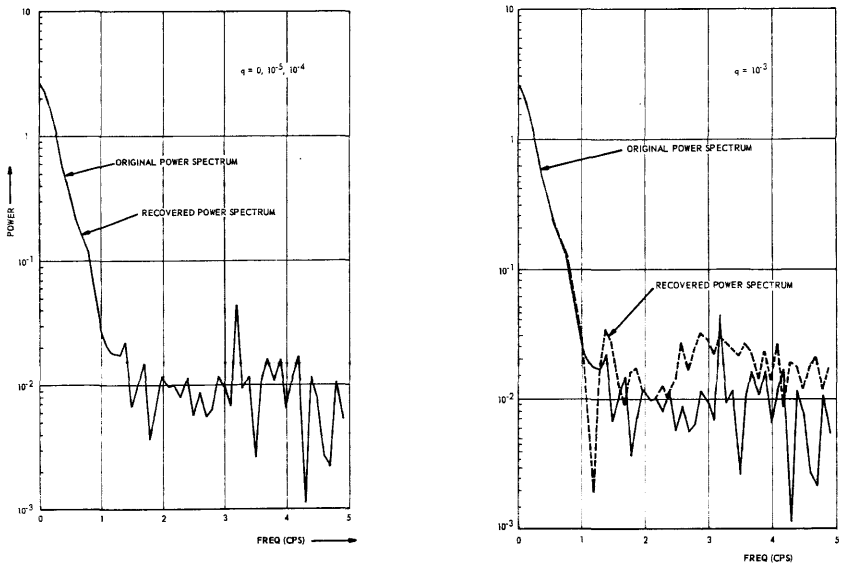


Figure 8—Original and noise-corrupted power spectra of an analog time function and the Rayleigh fundamental mode

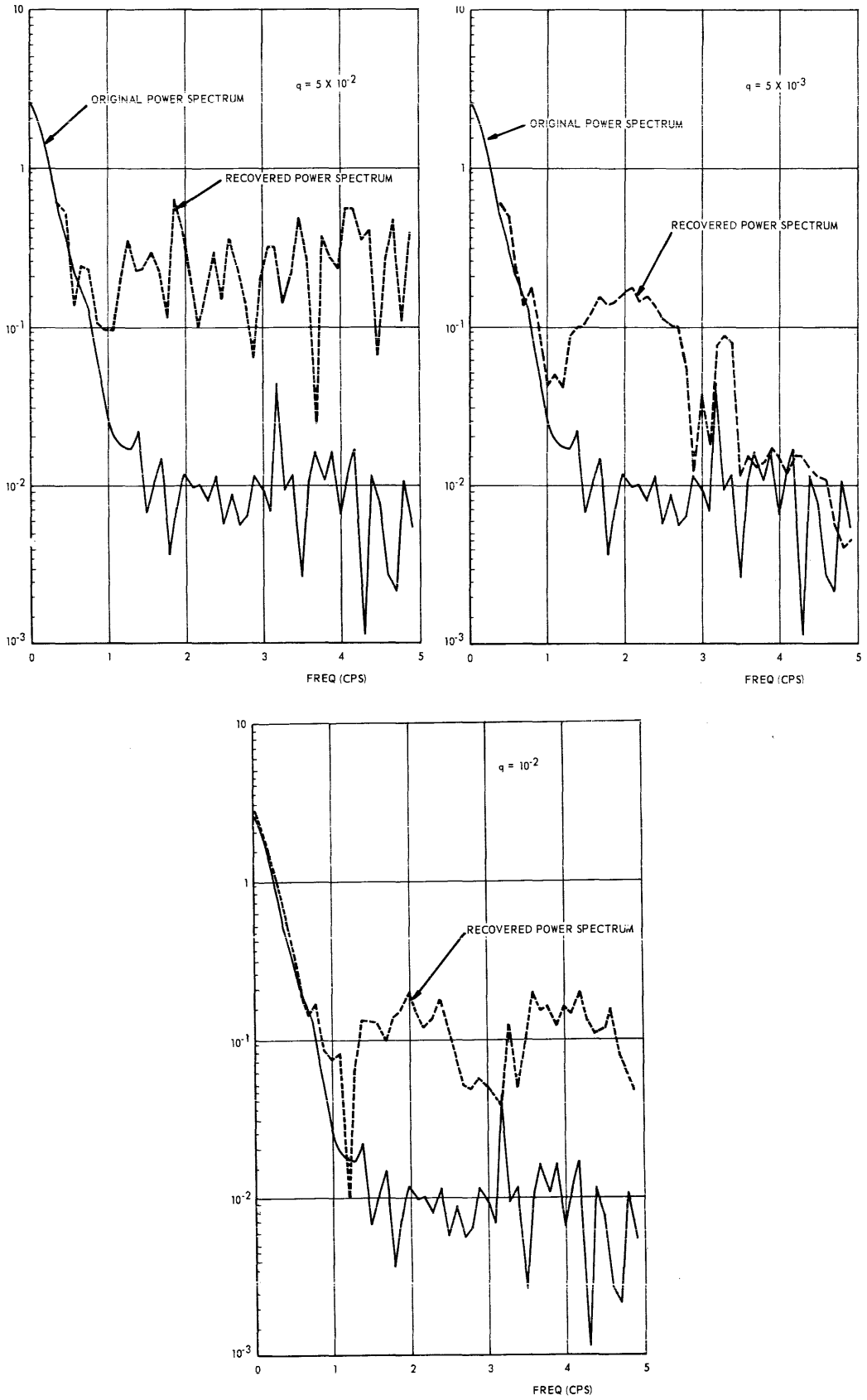


Figure 8 (Cont'd)

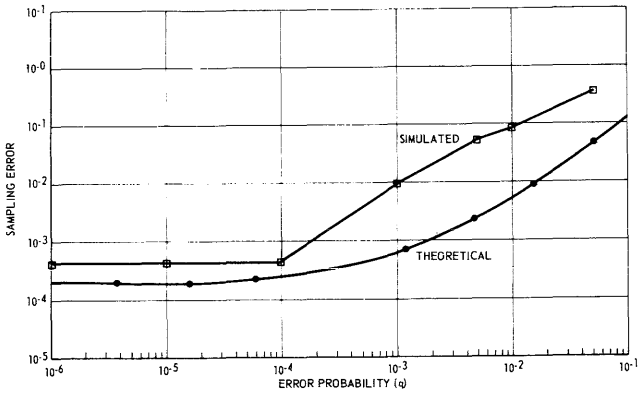


Figure 9—Sampling error versus channel error probability

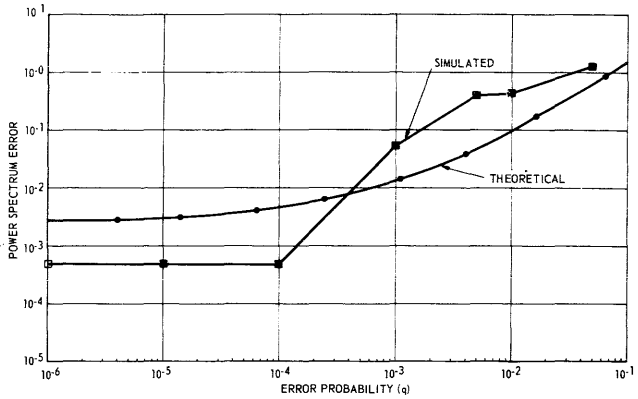


Figure 10—Power spectrum error versus channel error probability

Digital systems for array radar

by GEORGE A. CHAMPINE
UNIVAC Division, Sperry Rand Corporation
St. Paul, Minnesota

INTRODUCTION

The purpose of this paper is to present the role and philosophy of digital techniques, and especially of general purpose digital computers, in array radar systems.

Array radars and digital computers are extraordinarily well suited to each other. The flexibility and high data rate available from multibeam, multifunction array radars require an equally flexible, equally fast controller—a general purpose digital computer. The beam steering for many types of array radars is of a digital nature; and, of course, range, doppler frequency, and other quantities are inherently digital (after the information is extracted from the signal) just as in conventional radar.

The characteristics of digital computers that make them useful in a radar system are:

Perfect Memory: Digital memories can retain information with no loss in accuracy.

Time-sharing: The digital equipment is time-shared among many functions. This reduces system complexity and cost.

Decision-making ability: The logical design and digital nature of computers facilitates decision making.

Precision: Precision can be obtained as a linear function of cost.

Reliability: Digital computers can be among the most reliable subsystems of a radar.

A fully integrated digital controller and data processor can be implemented using a general purpose digital computer, in which all controller functions time-share the computer. This places a severe speed and reliability requirement on the computer, but off-the-shelf equipment is available to satisfy many present radar system requirements. The functions of a large scale array radar which can be integrated into the computer are:

- **Radar control**
  - Search raster generation
  - Beam steering
  - Range gating
  - Transmitter and receiver frequency
  - Transmitter and receiver beam width
  - Transmitter and receiver pulse type control
- **Radar data processing**
  - Detection rules
  - Track initiation
  - Range and angle tracking
  - Coasting
  - Target identification
  - Track-while-scan
- **Radar monitoring**
- **Display processing**
- **Console data processing**
- **Logistics**
- **Radar evaluation**
- **Radar checkout**

Typical system configuration

An example of a digital system for an array radar is shown in block diagram form in Figure 1. This diagram and most of the following discussion are a hypothetical composite drawn from several projects in current development.

In systems where great flexibility is required, and where events may be separated by time intervals shorter than the Input/Output transfer time, a Radar Instruction Buffer storage and timing unit has been used.

The computer sends ordered instructions (for the radar to execute) to the Radar Instruction Buffer; each instruction has a time tag to specify its time of execution. These instructions might be:

- Steer the transmitter to angular coordinates A and B.
- Send Pulse P_1 at time $Time_1$.
- Steer the track receiver beam to angular coordinates A and B.
- Range gate the signal at time T_2.

These instructions are held in the Radar Instruction Buffer, are decoded, and are gated to the proper equipment at the proper time to initiate the desired function.

The return signals are gathered by the receiver antenna and pass through the receiver, video con-

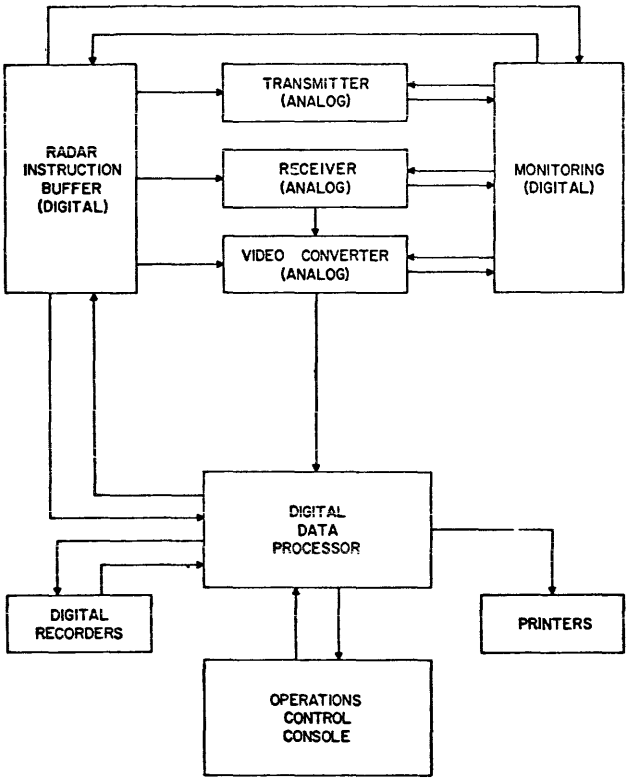


Figure 1 – Digital system for array radar functional block diagram

verter, and the analog-to-digital converter where they are encoded in digital format. The various parameters of these equipments are controlled by digital commands from the Radar Instruction Buffer.

The computer accesses test points in the radar via the monitoring equipment, and encodes the information for that test point in digital form for the computer. Provision is also made in the monitoring equipment for the computer to inject test signals when required for calibration and monitoring.

Manual control information from the Operations Control console is sent to the computer which translates the control information into radar instructions, adds timing information, and sends the instructions to the Radar Instruction Buffer. The computer also processes and formats information for display, and it can change the displayed information at the request of the operator.

Radar/computer interface

Figure 2 is a block diagram of the Radar Instruction Buffer. This approach to the Radar/Computer Interface has been used on large systems. On simpler array radars this equipment is omitted, and the various radar subsystems are tied directly to computer I/O channels.

Functions controlled

The functions listed in Table I are typical of those functions which might be controlled in a large array radar.

Table I – Functions controlled in a large array radar

Function	1 Transmitter	2 Receiver	3 Video converter
Beam steering angles	X	X	
Beam width	X	X	
Frequency	X	X	
Pulse time, duration	X	X	X
Pulse coding	X		X
Range gate position duration		X	X
Doppler filter range			X
Coherent integration			X
Calibration and monitoring	X	X	X

Method of control

The computer loads the Radar Instruction Buffer memory with a sequence of radar instructions. These instructions contain an address (equipment destination), time tag, and data. The instructions are loaded in time-ordered sequence. The Radar Instruction Buffer compares the time tag of the first instruction with its clock; when equality is reached, that instruction is sent to the designated address to initiate the function in real time. The next instruction is then loaded into the control

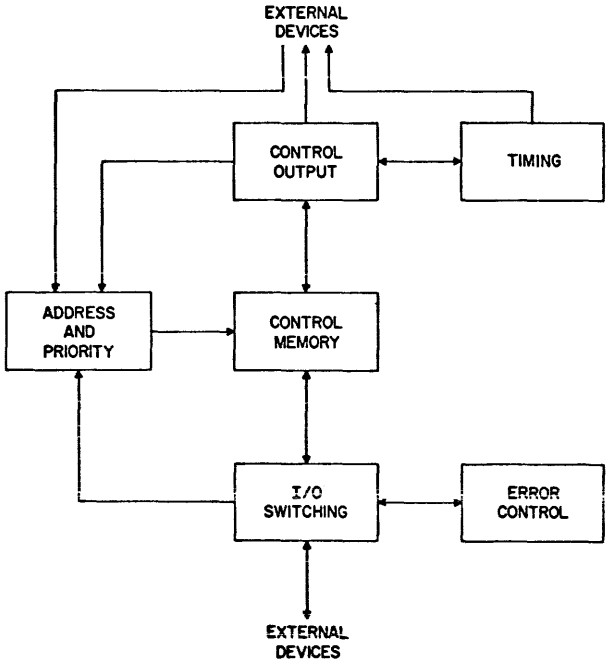


Figure 2 – Radar instruction buffer functional block diagram

output section where it is held until time equality is again reached.

The word length of instructions is determined by the number of events which must happen simultaneously, i.e., within the time to get one instruction from the Radar Instruction Buffer. For a multibeam tracking radar the word length must be quite large because of range gating and other signal processing requirements in the many beams. Many functions need not be critically timed except in very heavy target environments.

The monitoring information is requested through the Radar Instruction Buffer because of timing requirements. Some measurements must be made during dead time, some during active time; some require test signal injection and then read out at precisely controlled times.

Radar data processing

Radar data processing techniques for computer-controlled conventional radars have been thoroughly covered in the literature. The following discussion concerns the functions and the implementation rather than techniques.

Functions

The functions which have to be performed are:

- Search pattern generation
- Detection
- Track initiation
- Track-while-scan
- Range and angle tracking, smoothing, and extrapolation
- Automatic gain control
- Noise, sidelobe, and clutter rejection
- Video threshold determination
- Target identification
- Allocation of radar capacity

With a multifunction radar, all of these functions may have to be performed concurrently.

Implementation

The implementation of the above functions in an array radar is complicated by the flexibility of the radar. The use of the available pulses, time slots, and other radar capacities in an efficient manner requires a scheduling program which allocates the resources according to the target environment. A record must be kept of the allocation of pulses to targets so that as the returns are received the information can be sorted out, correlated with the proper tracks, or otherwise suitably processed. The result of this is that although the conventional basic computing algorithms are used for radar data processing, they require considerable additional program control logic in the array radar environment.

One of the largest uses of computer speed in radar data processing is the reduction of redundancy and randomness of radar signal tracking data, and this burden increases linearly with the tracking capacity of the system. The tasks which place the heaviest burden on the data processor are those done every cycle. Examples of these are:

- Range and angle tracking
- Pulse-to-pulse signal integration for detection
- Noise processing
- Automatic gain control
- Noise and sidelobe rejection

In a dense environment, even the fastest presently available stored-program computer may not have adequate speed. In this case, a wired-program processor may be used between the video converter and digital computer to perform the highly repetitive pulse-to-pulse correlation on the data. The correlation reduces the data rate (by reducing the redundancy) to a level where it can be handled by the stored-program computer. Development work is now proceeding to design advanced stored-program computers that are fast enough to process data from target densities substantially above present capability.

Radar monitoring

The large numbers of electronic components in array radars require a departure from manual methods of fault detection and diagnosis. The many repetitious circuits, which can be easily accessed by a computer, lend themselves to automated sampling and checking.

In an automated Radar Monitoring subsystem the computer would perform the following tasks:

(1) The computer sequences through all tests, sampling each test point at its required rate. The output of a signal channel is sampled relatively often to determine if the overall channel alignment is within limits. The component parts of a channel are sampled less often to determine if two components are drifting in compensating directions toward failure.

(2) The computer commands the appropriate test signals, if required, via the Radar Instruction Buffer during the proper part of the radar cycle.

(3) The computer examines the results of the test and takes appropriate action. If analysis of the failure indicates that it can be remedied by computer action, the computer performs the programmed corrective measures. If not, the maintenance man is notified by a message output.

If the failure cannot be localized to one least replaceable unit, further tests are performed, and the results are associatively processed to localize the failure to a least replaceable unit.

(4) The computer supplies instructions to the maintenance personnel. The radar is so complex that available maintenance personnel will be limited in their ability to diagnose failures or to take proper corrective action.
(5) The computer services maintenance personnel messages to the computer. These include:
- A request for special test so the maintenance man can locally adjust components or diagnose failures with test equipment.
- A notice that a failed component has been replaced or adjusted.
- A list of all components likely to fail in the next time period.
- A list of all components that exceed a certain margin.

(6) The computer assists the maintenance personnel in preventive maintenance by notification when timely replacement of components is required, or failure is imminent.
(7) The computer records component failure and replacement information for reliability and logistics purposes. The current failure information is used to estimate the degradation of radar performance; this information is provided to the Maintenance Console and Operations Control Console.

Display and control

Although computer-controlled radars are capable of fully automatic operation, displays are necessary for the radar operator to monitor the system operation. In addition, controls must be available to provide manual control of the radar. In conventional radars the received signals can be displayed directly on a cathode ray tube because the beam is moved in a regular pattern. In an array radar the time-sharing of the beam among many functions requires that display and control be carried out through the computer.

In dense target environments the computer distributes the information to several operators according to task and work load. Since computer-driven displays are largely general-purpose, the equipment can be designed or obtained before its precise method of use has been decided. Computer-driven displays can provide summarized or derived quantities which are much more useful than raw data. Examples of these quantities are:

- Bearing angle
- Impact point
- Distance of closest approach
- Radar cross-section

The following display and control description illustrates the requirements of an array radar system.

Operations control console

The Operations Control Console has two functions:
(1) to display sufficient information to keep the operator informed of the radar operational status and performance, and
(2) to enable the operator to control the radar to its fullest extent, where desired, through a complete set of controls.

Input and output signals, with the exception of the raw video displays, are via the computer. Operation is automatic, and the operator need not intervene unless he wants to.

1. Displays

The displays include:
- A-scope
- Azimuth vs. elevation digital CRT display
- Alphanumeric digital CRT display
- Azimuth vs. range digital CRT display
- Special purpose status indications
- Medium speed printer

The A-scope is used to present targets in a beam, but not necessarily in track, and can be manually switched to any beam. The A-scope is triggered by the computer, but the trigger can be modified in position and velocity from the console.

The azimuth vs. elevation scope has characters displayed on it by the computer to show the location and status of tracks.

The azimuth vs. range scope similarly has characters displayed on it by the computer to show the location and status of tracks.

The alphanumeric scope is used to display operational data (for any target designated by the operator) such as:

- Track status
- Position, velocity
- Target identity
- Altitude
- Distance to nearest object
- Signal to noise ratio

It also displays information, for the radar as a whole, such as:
- Number of objects in track-while-scan
- Number of objects in precision track
- Per cent of traffic capability being used

2. Controls

The Operations Control Console has the following controls:
- Mode control (auto, trim, manual)
- Cursor/light gun for each CRT display
- Range and range rate trim, scale adjustment, and beam selector on A-scope

- Function designator – designate primary track, delete track, change to track-while-scan, change to precision track
- Keyboard – for typing control messages to computer
- Special controls – false alarm rate, noise sample control, steering angle controls, range control
- Search boundaries in range, azimuth, elevation

3. Operation

The radar system has three modes of operation: automatic, trim, and manual. In automatic, the operator can request information and alternate procedures, but has no direct control over the radar. For example, the operator could request to have displayed numerically the position, velocity, time of arrival, radar cross section, and orbital parameters of a target. Examples of other requests he could make are:

- To see video on a particular track.
- That a particular target be put in precision track.
- That the false alarm threshold be changed.
- That the search boundaries be changed.

In the trim mode the operator has all functions of automatic mode, and in addition can manually modify the steering of the beam and the range gates. This mode is used to let the operator assist the radar in abnormal situations.

In the manual mode the operator assumes complete control and must supply all parameters to the radar. The computer protects the radar from damage by operator error. This mode is used for checkout, special test, and diagnosis of faults.

The cursor or light gun allows the operator to designate targets for action or information.

Monitoring console

The Monitoring console has two functions:

(1) to display overall system status and summary information to the maintenance supervisor, and
(2) to allow the request of special diagnostic and test programs and information on the equipment status of the radar.

1. Display

The Monitoring Console will display information concerning the transmitter, receiver, and video converter. It will consist of:

- General purpose CRT
- Keyboard
- Medium speed printer

A high speed printer will be available for printing large amounts of data.

2. Operation

The operator may request the following displays on the CRT and/or printer by typing in the appropriate message on the keyboard. In the following, "component" refers to the smallest testable unit.

- Display in their proper spatial relation all transmitting or receiving antenna elements that are out of tolerance.
- Display (list) any one of the items in Table II.

In addition, the operator will be able to request special radar tests and results via the keyboard.

Table II – Displays for equipment monitoring

Display quantity	1 Transm	2 Receiver	3 Video converter
Per cent of channels out	X	X	
First moment of outages	X	X	
Peak power (sensitivity) available	X	(X)	
Function outages	X	X	X
Redundant components out	X	X	X
Estimated angular accuracy		X	
Estimated statistical sidelobe level	X	X	X
All failed or out-of-tolerance components	X	X	X
Signal path with poorest performance	X	X	X
Component with least margin or adjustment	X	X	X
All signal channels within X of failing	X	X	X
All components within X of failing	X	X	X
Histograms of component tolerances	X	X	X
Histograms of channel tolerances	X	X	X
Histogram of percent adjusting remaining	X	X	X
All failed channels and components	X	X	X
All components requiring manual adjustment	X	X	X
All components at limit of automatic adjustment	X	X	X

Logistics

A large scale array radar system may have spare parts inventories of several thousand items. Many of

these items have long lead times and require that the reordering point be determined as a function of the usage rate and required lead time. The computer has most of the data required to maintain spare parts inventory records from the monitoring and parts replacement information. Since the computer is rarely in a full-load condition, the reserve computing capacity can be used to maintain logistics information on a low priority basis at little extra cost. The automating of this function will reduce the manpower requirements and cost for maintenance.

Functions

The following logistic functions would be performed:

- Maintain spare parts inventory.
  Issue reorder requests based on lead time and usage rate for each item.
  Maintain reliability records for failure rate, drive rate, frequency of adjustment required, mean time to failure, and mean time to repair.

Implementation

In normal monitoring operation, all manual adustments, failures, and parts replacement information are recorded on magnetic tape. New supplies from the manufacturer or repair depot are recorded on punched cards, and this information, together with the monitoring data, is used to update the spare parts inventory periodically.

Where applicable, drift rates are computed from monitoring information and used to modify the reorder point and to predict end-of-life failures.

The statistical summaries on drift rate, time to failure, time to adjustment for each component and the system are also derived from the monitoring data.

Simulation

Real time simulation of operational situations can be performed by the central computer. The simulation can be done with artificial data generated within the computer, or it can be done by replaying digitally recorded data from real missions. This replay capability is very important in post mission analysis where considerably more information is recorded than can be displayed in real time. During the simulation, the displays and various other equipment are activated, depending on the purpose.

Simulations which involve the displays are used to train operators and to maintain their proficiency in situations encountered infrequently. It also simulates malfunctions to exercise maintenance personnel.

Simulations also provide realistic data to check and evaluate parts of the system equipment and programs.

Checkout and evaluation

The computer can be very useful in the checkout and evaluation of each subsystem before it goes into the radar, and of the entire radar system once it is operational. Any complex piece of equipment requires a formal checkout procedure to verify its correct operation. If these checkout procedures are implemented on a computer during the manufacture of the equipment, then the diagnosis and location of faults can be done quickly. These checkout programs are very similar to the usual fault detection and diagnosis programs that are supplied to maintain equipment, and can often be obtained for little additional cost.

CONCLUSION

The preceding paragraphs have presented the role of general purpose digital computers in array systems. The topics discussed were:

- Advantages of digital computers
- Data processor system configuration
- Control of radar
- Radar data processing
- Radar equipment monitoring
- Radar-computer interface
- Computer support for display and control
- Computer support of logistics
- Simulation of system operation
- Program organization
- Checkout and evaluation assisted by computer

Computers have proven themselves to be highly useful members of array radar systems in performing the tasks assigned to them. As the capability of digital data systems equipment and programming continues to increase, it is reasonable to expect that the role of computers in array radars will continue to increase.

DIAMAG: a multi-access system for on-line Algol programming

by L. BOLLIET, A. AUROUX, and J. BELLINO
Institut de Mathématiques Appliquées, Université de Grenoble
Saint-Martin-d'Hères, France

INTRODUCTION

On-line programming

On-line programming is characterized by a complete man-machine interaction at the level of *program construction* as well as at the level of *program evaluation*. In other words, there should not be any difference between syntactic evaluation and semantic evaluation in on-line programming.

The detection and correction of syntactic errors are performed as the source program is read and analyzed (syntactic evaluation phase). The user can make changes, deletions, and/or additions to his program easily, since he is able to communicate with his program at the source language level.

The detection and the correction of semantic errors, which entails execution, must also be performed in much the same way; that is, by the use of source program symbolic references for the designation of quantities at execution time.

This possibility gives the programmer a direct communication with the program being evaluated. He should be able to

- read the value of a variable (dynamic reading);
- change the value of a variable (dynamic assignment);
- stop and restart the execution (dynamic control), and
- restart the execution at a given point (dynamic "go to").

Conversational compilation and execution

Conversational compilation and program construction

There are two basic ways of constructing programs.

(1) *Recursive construction:* construction of a program in a single phase starting from the outer block and proceeding to the inner blocks.

This is the usual way and, so to say, the only way of writing Algol programs when using batch processing systems.

(2) *Iterative construction:* construction of a program in as many phases as there are elementary blocks in the program.

Each block is constructed, compiled and evaluated independently. The whole program can be built up with these elementary blocks. With conversational systems this method seems to be of great importance, since separately constructed and compiled blocks may be shared by several users.

Both methods should be made available to the users of a multi-access system which combines to a significant extent conversational and batch processing compilation as well as complete compatibility of programs in either mode.

Conversational compilation is characterized by

- the use of special commands to erase, change, insert statements within the program being constructed, and
- the use of incremental techniques.

Conversational execution and program evaluation

The same principles apply to program evaluation.

(1) *Recursive evaluation:* evaluation in a single phase. This is also the only way of evaluating programs in a batch processing environment.

(2) *Iterative evaluation:* After a small block of program has been constructed, it can be evaluated independently provided that values be given to non-local quantities. The result of the evaluation, i.e. the new values of non-local quantities, can be used later on for the evaluation of larger blocks.

Again, in conversational systems both methods should be available to give a greater flexibility to the programmers.

Conversational execution is characterized by

- the use of special commands to ask for or to transmit information to the program being evalu-

ated as mentioned before;

- the use of the source symbolic references to designate the quantities.

The use of incremental techniques in conversational systems

The need for incremental compilation, i.e. independent and separate compilation of each program statement, is apparent for conversational systems. Since a large part of the task consists in erasing old statements, changing old statements into new ones and inserting new statements, incremental compilation avoids complete recompilation of the program so far constructed each time a change is made, either freely by the programmer or required by the system after error detection. In this system, incremental compilation is used at two different levels:

(1) *Statement level* for recursive program construction;

(2) *Block level* for iterative program construction and access to a common set of already constructed programs.

The reason for these two levels is the following: Admittedly, there is no difference between block and statement from a syntactic point of view, but a block can be considered as a self-sufficient piece of program but not an ordinary statement. The block inclusion mechanism is described later on.

Incremental compilation at the statement level does not differ from similar methods for Fortran[1] and Algol.[2] The program structure is represented as a list structure which mirrors the logical sequencing of statements (as opposed to the physical sequencing in the computer storage).

Incremental compilation at the block level proceeds as follows: Blocks already constructed are stored both in source and object forms in a list-structured area within the filing system. When one of these blocks is to be inserted within a block being constructed, there is no physical insertion (i.e. recopying) but rather insertion of pointers to the enclosed block. The object code for blocks has to be reentrant since it can be used by more than one program at a time. Incremental compilation of blocks is thus achieved through independent compilation of inner blocks using the following features:

(a) *Program library capabilities.* Any elementary block of a program can be compiled and stored in the file system for later use. Such a block of program consists of a source text and an object text.

(b) *References to non-local quantities.* If a block is to be compiled independently, means should be provided for referencing identifiers declared outside this block.

(c) *Block nesting and equivalence of references.* In the process of iterative construction, larger blocks have to be built using smaller blocks already constructed. If a smaller block using non-local identifiers is to be incorporated into a larger block within which the non-local identifiers of the smaller block are local, means should be provided to establish an equivalence of identifiers.

Concurrent use of conversational and batch processing modes

In a multi-access system, batch processing and conversational processing are not to be considered as exclusive of each other since these modes are in fact complementary and are to be used concurrently at various stages of program construction and evaluation.

The system described here is based on two important features:

(1) availability of two different compilers and interpreters for Algol;

(2) compatibility between batch processing and conversational compliation and execution.

Once a program has been constructed and debugged through conversational techniques, it has to be compiled and executed using batch processing techniques if it is to be run a number of times.

However, the compatibility between the two processing modes is one-way: conversational toward batch, i.e. if the conversational form of a program (both source and object forms) has been destroyed, it cannot be reconstructed from the batch processing form.

The transformation mechanism is described below.

System description and organization

Programming language and command language

The programming language used is Algol 60, augmented by a number of special instructions which constitute the command language itself.

The command language is for execution of operations on quantities of the type *file*. A *file* is a sequence of elements of the four following types: bits, characters, machine-words, lines. (A line is a grouping in a given format of one or more of the following types: bit, character, machine-word.)

In general, *identifiers* are used to designate files, and *pointers* (integer numbers), to indicate elements within these files.

There are three basic operations on files:

(1) *Insertion*—insert (element, pointer, file) which allows insertion within "file" at the point defined by "pointer" of another file designated by "element."

(2) *Extraction*—extract (pointer 1, pointer 2, file) for extracting within "file" the piece of file between "pointer 1" and "pointer 2."

(3) *Concatenation*—concatenate (file 1, file 2) for the concatenation of the two files: "file 1" and "file 2."

Utilization modes

There are three modes (or levels) of utilization:

- system mode,
- compiler mode (with two submodes: conversational and batch), and
- interpreter mode (with two submodes: program and desk calculator).

The command language is defined syntactically as follows (*EOC* stands for *E*nd *O*f *C*ommand delimiter):

```
        <activity> ::=<library activity> | <user
                      activity>
<library activity> ::=begin EOC
                      COMMON (<password>)
                      EOC
                      <job list> EOC
                      end EOC
   <user activity> ::=begin EOC
                      NAME (<user name>,
                      <password>) EOC
                      <job list> EOC
                      end EOC
        <job list> ::=<job description> |
                      <job list> EOC <job
                      description>
<job description> ::=FILE (<file-name>) EOC
                      <command list> |
                      COMMON FILE (<file-
                      name>) EOC
                      <command list>
 <command list> ::=<command> |
                      <command list> EOC
                      <command>
     <command> ::=<filing command> |
                      <compiling command> |
                      <interpreting command>
```

Access to the system is made through the *begin* command (log on command), and disconnection from the system, through the *end* command (log off command).

The NAME command creates an activity labelled with the user's name. During an activity, a user can perform a sequence of jobs, each of them labelled with a <file-name> which characterizes a particular job.

System mode and filing commands

This mode has the highest hierarchy; that is, entry and exit from the Multi-Access System always take place at this level.

In addition to the *begin*, *end*, NAME and FILE commands, the system mode has a number of filing commands described as follows:

```
<filing command> ::= CONCAT (<file-name>) |
                     INSERT (<pointer>, <file-
                            name>) |
                     EXTRACT (<pointer 1>,
                            <pointer 2>,
                            <file-name>,
                            <file-mode>) |
                     LIST (<pointer 1<,
                          <pointer 2>) |
                     ERASE (<pointer 1>,
                           <pointer 2>) |
                     COPY (<file-name>,
                           <file-mode>) |
                     COPY (<file-name>,
                           <user-name>,
                           <file-mode>) |
                     READ (<external de-
                           vice>) |
                     WRITE (<external de-
                           vice>) |
                     STORE (<file-mode>) |
                     ESTORE |
                     DELETE |
                     EXEC (INFILE (<file-name
                           list>),
                           OUTFILE (<file
                           name list>))
```

Changes in common library files can only be made through a library activity, whereas normal operations (LIST, EXTRACT, COPY, EXEC) are carried out through user activity. The file-mode specification gives the utilization mode allowed for this file: RO (Read Only) and WR (Write and Read). The three commands ERASE, CONCAT and INSERT cannot be used with read only files as first operand.

Within a library activity, only filing commands (except EXEC) can be used. The filing commands are for execution of elementary operations on files with the following rules:

For unary operations: The operand and the result file have the same designation, defined as the file-name of the corresponding job description. The five commands for unary operations are:

LIST	Listing of the part of the operand file between <pointer 1> and <pointer 2>.
ERASE	Deleting of the part of the operand file between <pointer 1> and<pointer 2>.
STORE	The information which follows this command is considered as a file and stored in the system with the designation defined for the operand.

ESTORE Flags the end of the file which follows the STORE command.

DELETE Deletes the operand file from the file system.

The READ (WRITE) operation permits direct input (output) of a file from (to) external media.

For binary operations: The first operand file and the result file have the same designation, defined as the file-name of the corresponding job description, except for the EXTRACT and COPY commands. The second operand file is defined as the last parameter of the related command. The commands for binary operations are:

CONCAT The first operand file is concatenated with the parameter file. The parameter file is unchanged and the first operand file is modified.

INSERT Inserts inside the first operand file, the parameter file from the point defined by <pointer>. The parameter file is unchanged and the first operand file is modified.

EXTRACT Extracts from the first operand file the part between <pointer 1> and <pointer 2> which becomes the parameter file.

COPY A particular case of the EXTRACT command which allows duplication of the first operand file designated by <file-name> which takes as name the parameter file.

The following command has a variable number of operands:

EXEC Allows the execution of a program whose name is defined in the corresponding job description and using as input files and output files the associated file-name lists.

Compiler mode and compiling commands

Compiling commands are defined as follows:

<compiling command> ::= COMPILE (<language name>, <file-name>) | ENDCOMP | SAVECOMP | RESCOMP | REPLACE (<syntactic number>) | TRANSFORM (<file-name>) | EXTERNAL (<specification part>) | EQUIV (<identifier pair list>) | INCLUDE (<file-name 1>, <file-name 2>)

<identifier pair list> ::= <identifier> : <identifier> | <identifier pair list>, <identifier> : <identifier>

In the compiler mode, there are two submodes:

- Batch processing mode
- Conversational mode

(a) *Batch processing mode*—The command will

COMPILE (<language name>, <file-name>)

will cause the compilation of the source program designated by <file-name> and written in the language defined by <language name>. The result of the compilation (object program) is the file defined in the corresponding job description. The three languages Algol B ("B" for *B*atch), Cobol and Fortran IV are available in the batch processing mode.

The end of a compilation must be denoted by the ENDCOMP command.

(b) *Conversational mode*—There is only one language, Algol C ("C" for *C*onversational), available in conversational mode. The source program is to be written just after the COMPILE command. It will be stored in the file designated in the COMPILE command. The object program is the file defined in the corresponding job description. The conversational Algol compiler is organized as follows:

The syntactic evaluation is performed with a sequential algorithm using a single stack (syntactic stack) and the object text is generated at the same time.

Besides the source text and object text files, the following files are used during the compilation:

- syntactic stack,
- working storage for compiler state, and
- symbolic references table.

All the references to the symbolic quantities are made indirectly through a symbolic references table built for each block level. At the closing of a block, the part of the symbolic references table corresponding to this level is transferred into the object program.

In this mode the compilation can be interrupted at any time. It is possible to save the current state of the compiler with the SAVECOMP command which puts away the files associated with the compilation. This state can be restored through the RESCOMP command.

Syntactic numbering and incremental compliation of statements

In order to facilitate correction, a syntactic numbering of statements is performed while the program is being typed at the terminal. As the source program

is read, character by character, a syntactic analysis is carried out at the statement level so as to determine the correct syntactic number of each statement. Since a statement in Algol is a recursively defined concept, this syntactic number is not a single integer but a sequence of integers separated by periods which reflects the structure of the statement. For instance, in the following program syntactic numbers are written in front of the corresponding statements.

```
1 begin 1.1 A:=B+C;
        1.2 begin
            1.2.1 X=Y+Z;
            1.2.2 if I<J then
                        1.2.2.1 goto E
                        else
                        1.2.2.2. I:=J-K
            end;
        1.3 P(X,Y,Z)
end
```

In this syntactic numbering, the declaration list is treated as a compound statement, though declarations cannot be compiled quite as independently as ordinary statements. Specification part and value part in procedure declarations are treated similarly.

For program modification there is only one replacement rule which can be stated as follows: a statement can only be replaced by another statement. Thus, erasing a statement amounts to replacing it by an empty statement, changing a statement into several statements amounts to replacing it by a compound statement. When a statement (or several statements) is to be inserted into two other statements, a compound statement consisting of either one of the two statements and the statement to be inserted must be created. With this rule, a correct syntactic numbering can be maintained throughout the program construction phase.

The REPLACE command only specifies the syntactic number of the statement of this command, the syntactic number will be printed out, thus permitting the programmer to type the new statement. If the newly specified statement contains other statements without modifications, one can use the syntactic number of those statements enclosed between parentheses instead of recopying the statements. For instance, in the preceding example, if we want to replace the statement 1.2.2. without changing the statements 1.2.2.1. and 1.2.2.2., we can write:

1.2.2. *if* I≥J *then* (1.2.2.1.) *else* (1.2.2.2.)

These replacements are freely performed by the programmer or requested from the compiler itself when an error occurs.

Block inclusion mechanism

One of the most important compiling commands for program construction is the INCLUDE command which permits inclusion of an already constructed block within a block being constructed. The two file-name parameters of this command designate the source program file and the object program file to be included within the corresponding files of the block under construction. We have previously mentioned that there is no physical inclusion (i.e. recopying) but, rather, logical inclusion (i.e. appropriate positioning of pointers). In the source text, the INCLUDE command is considered as a statement and given a syntactic number as any statement for possible replacement. In the object text, the INCLUDE command will cause the compliation of a call to a re-entrant subroutine since the included block may be used in more than one program.

Program transformation from conversational mode to batch processing mode

This transformation is performed in two stages:

(1) Transformation of the source program written for conversational mode into a source program for batch processing mode;

(2) Compilation of the new source program using the batch processing compiler.

In the first stage, the EXTERNAL, EQUIV and INCLUDE commands and STOP statements are eliminated from the source text, the source text being updated accordingly. In this transformation INCLUDE commands can be considered as text-editing macros, i.e. they will be replaced with the corresponding source text. This process is recursive since the newly inserted piece of source text may in turn contain INCLUDE commands. The EXTERNAL and EQUIV commands are eliminated after the correct replacing of external identifiers according to the specifications given in the EQUIV command.

More details about the writing of programs and the implementation scheme are given in reference 3.

Interpreter mode and interpreting commands

Interpreting commands are defined as follows:

```
<interpreting command> ::= INTER (INFILE
                               (<file-name list>),
                               OUTFILE (<file-
                               name list>)) |
                           INITIAL (<name-
                           value pair list>) |
                           VALUE (<variable
                           list>) |
                           <variable> := <ex-
                           pression> |
```

STOP |
START |
go to <label> |
ENDINT |
SAVEINT |
RESINT |

<name-value pair list> ::= <name value pair> |
<name value pair list>, <name value pair>

<name-value pair> ::= <arithmetic variable> : <number> |
<Boolean variable> : <Boolean constant> |
<array identifier> : (<list of values>) |
<procedure identifier> : (<file-name>, <list of initial values for external quantities>)

The execution is performed using a single stack (run-time stack).

The INTER command causes the execution of the object program designated in the job description using as data the files designated in the two file-name lists.

An execution can be interrupted in three ways:

- The end of the object program.
- A STOP instruction written by the programmer within the Algol source program.
- A STOP command from the terminal which has originated the execution in progress.

A STOP instruction will suspend the execution of a program at the program point flagged with the STOP instruction. A STOP command will suspend the execution of a program at an unpredictable point depending upon the time this command is obeyed. The STOP command is to be used when a program loops or produces incorrect results. Whenever a program has been interrupted through a STOP instruction or a STOP command, it is possible to resume the execution with the START command.

After any interruption, the results of the execution so far performed can be checked by using the following three special interpretation commands:

(1) VALUE (<variable list>) will print the actual value of the variables given in the <variable list>

(2) <variable> := <expression> will change the current value of the <variable> to that of the <expression>

(3) *go to* <label> will transfer the execution to the program point defined by <label> if this is legal. If it is an illegal or unknown label, an error message will be printed out.

In these three commands, all quantities are designated by their symbolic references in the source program. The state of the interpretation can be saved with the SAVEINT command, which stores the files associated with this interpretation (run-time stack, working storage for interpreter state). This state can be restored with the RESINT command.

Exit from the interpreter mode is performed through the ENDINT command.

Desk calculator mode

One enters the interpreter mode without an object program associated with the file-name of the FILE command in the job description head and without input and output file name in the INTER command. Only the two interpreting commands

<variable> := <expression>
VALUE (<variable list>)

can be used.

In that case, these two commands are similar to instructions executed immediately and correspond to a desk calculator mode.

CONCLUSION

The system described above allows both interactive construction and evaluation of Algol programs in a Multi-Access System and batch processing of programs written in other languages. This system is being implemented on an IBM 7044 computer with a 1301 Disk Unit and a PDP-8 satellite computer serving a large number of teletypes. A preliminary version of this system, including a small conversational Algol compiler using an IBM 1401 as satellite computer and two IBM 1050 terminals was implemented and has been operational since June, 1966.[4]

REFERENCES

1 K LOCK
Structuring programs for multiprogram time-sharing, on-line applications
FJCC 27 457-472 1965

2 J M KELLER *et al*
Remote computing an experimental system part 2: internal design
SJCC Proceedings 426-443 1964

3 L BOLLIET J LE PALMEC
Compilateurs incrémentiels: caractéristiques techniques et méthodes d'implantation
Note technique IMAG to be published

4 A AUROUX J BELLINO
Système en temps partagé 1401/7044 en mode moniteur et mode conversationnel
Colloque AFIRO sur le traitement à distance et l'utilisation des calculateurs en temps réel et en temps partagé Grenoble to be published by Dunod Paris 31 mai-3 juin 1966

GRAF:Graphic Additions to FORTRAN

by A. HURWITZ and J. P. CITRON
IBM Los Angeles Scientific Center
Los Angeles, California

and

J. B. YEATON*
Health Sciences Computing Facility, UCLA
Los Angeles, California

INTRODUCTION

With the growing widespread use of graphic display devices, higher level graphic display languages which are easy to use are a necessity. Many systems have been developed which are designed around the use of online graphic display terminals. These systems generally have their own language and are designed to run in an environment dedicated to the given system.

On the other hand, means have been developed to allow a higher level language programmer to address graphic devices such as microfilm recorders, incremental potters, or cathode ray tube displays, through the use of pre-packaged subroutines entered by means of CALL statements. This represents no real extension of the language.

GRAF is intended to fill a gap between the two extremes of a package of subroutines in FORTRAN on one hand and a completely dedicated graphical display system on the other. In GRAF, new statements are added to the FORTRAN language. These statements are designed to be as consistent as possible with FORTRAN. The advantages of this design are that GRAF will be easier to implement, easier to teach to FORTRAN programmers, and easier to read.

GRAF was designed with a particular version of FORTRAN and a particular display device in mind. However, the features of GRAF are to a great degree machine independent and modular so that it can be adapted to different configurations.

GRAF was designed to extend OS/360 FORTRAN IV (E Level subset) and to operate with the IBM 2250 Model I display device with a buffer, absolute vector feature, light pen, program function keyboard, and alphameric keyboard.

*Now with University of California School of Medicine at San Francisco.

IMAGE GENERATION

Display variable

The central nation in GRAF is that of a *display variable*. This is a new type of FORTRAN variable whose value is actually a string of graphic device orders capable of generating directly a display of points, lines, and characters when transmitted to a display device.

Display variables are similar to ordinary FORTRAN variables in many ways. The same rules for naming them apply. They are declared in a declaration statement and can be dimensioned. They can appear in EQUIVALENCE and COMMON statements, and they can be passed to subroutines as arguments. Further, they can be assigned values by an assignment statement similar to the arithmetic assignment statement in FORTRAN.

Notational conventions

In describing the GRAF statements, we will use the following conventions: dv, dv_1, dv_2,..., dv_n will represent any display variables, lower case letters will represent names to be replaced by the programmer—the first letter indicating the type (integer or real) if it is not a display variable, upper case letters will represent names which may not be changed by the programmer. The names "dx" and "df" refer to display expressions and display functions respectively. Items enclosed in brackets are optional and items

written in a column indicate a choice among those items is to be made.

All functions except display functions follow the usual FORTRAN rules as to type.

DISPLAY declaration

Display variables must be declared by using the DISPLAY declaration. Its form is

DISPLAY $dv_1[(k_1)],dv_2[(k_2)],..., dv_n[(k_n)]$

where n is greater than zero and for each dv_j for j = 1,2, ..., n dv_j is declared to be a display variable or an array of display variables whose dimensions are specified by k_j where k_j is composed of one to three unsigned integer constants. Each element of an array of display variables can be treated independently just as ordinary subscripted FORTRAN variables can be treated independently of each other.

For example,

DISPLAY A, J, Q(17), TR(2, 7, 5)

defines A and J to be display variables, and Q and TR to be arrays of display variables.

Display function

A *display function* is a FORTRAN function whose value is a string of graphic orders. The built-in display functions of GRAF are:

POINT (x,y) which generates orders for plotting a point

LINE (x,y) which generates orders for plotting a line

PLACE (x,y) which generates orders to change beam position without plotting

CHAR (string, length, mode) which generates orders for plotting a string of characters

PRINT n, list which creates a string of characters using a list and a format almost exactly like the PRINT of FORTRAN and then generates the orders necessary to plot the resulting string of characters.

The FORMAT statement used by the PRINT function has an expanded set of "carriage control characters" which control the size and protection status of characters and insertion of cursors.

Display expression

A *display expression* is a sequence of display variables and display functions separated by plus signs. The value of a display expression is the string of graphic orders which is the concatenation of the values of the display variables and display functions in the order from left to right. Its form is:

$$dv\left[+\,dv\right]\left[+\,dv\right]\cdots$$

For example, if A and B are display variables, then:

A+POINT(O, YW7)+B

is a display expression whose value is the string of graphic orders of A following by the orders generated by POINT followed by the orders generated by B.

Display assignment statement

The *display assignment* statement is used to assign a value to a display variable. Its form is:

$$dv\,[(k)] = \frac{dv}{df}\left[+\frac{dv}{df}\right]\left[+\frac{dv}{df}\right]\cdots$$

where k, if present, specifies the subscript or subscripts of dv.

For example, the following are display assignment statements, assuming the variables A, B, SQU, POLE, K99 appeared in the DISPLAY statement below:

```
DISPLAY A, B, SQU, POLE(11), K99 (3, 2, 4)
A = B
A = A + B
A = B + A
SQU = POLE(5) + POLE(3)
POLE(1) = PLACE(RX,0)
POLE(1) = PLACE(RX, O) + LINE(X3, Y7)
K99 = PLACE(0, 0) + PRINT 14, (ZK(I), I =
  1, 7) + PLACE(2000,2000)
```

Coordinate specification

Subroutines are provided to specify a coordinate transformation so that the built-in functions PLACE, POINT, and LINE which use x, y coordinates will be able to make the transformation from user coordinates to device coordinates. If no specification is made, device coordinates will be used throughout.

Resetting a display variable

In order to set a display variable to the empty string, the RESET subroutine is used. Its form is:

CALL RESET ($dv_1, dv_2, \ldots, dv_n$)

OUTPUT OF A DISPLAY VARIABLE

The transmission of display variables to the display device and the subsequent erasure and replotting are controlled by the FORTRAN functions PLOT, UNPLOT, ERASE, and the subroutine BLANK.

PLOT

The form of the PLOT functions is:

PLOT ($dv_1, dv_2, \ldots, dv_n$)

The function transmits the values of its arguments, that is, the strings of graphic orders corresponding to the display variables, to the device buffer if there is enough storage available in the buffer. If not enough storage is available for all the strings then no transmission takes place.

The numerical value returned by PLOT is positive if transmission took place and negative otherwise; its absolute value is equal to the amount of available buffer storage.

If any of the display variables are dimensioned and are not subscripted, then the entire array is plotted.

Each time a display variable is plotted, a new *instance* is said to be created for that display variable. The newly created instance becomes the *current* instance. As many instances as desired may be created for a display variable and the value of the variable may be changed between the uses of PLOT. For example, the following sequence of statements creates three instances of A, two of which have the same value:

```
A = PLACE(0, 0) + LINE(0, 4095)
T = PLOT(A, A)
A = PRINT 18, U, V, W
T = PLOT(A)
```

UNPLOT

The FORTRAN function UNPLOT does exactly the opposite of PLOT. It removes the current instance (if there is one) of a display variable and makes the previous current instance (if there is one) the current instance. Its form is:

$$\text{UNPLOT}(dv_1, dv_2, \ldots, dv_n)$$

The numerical value returned is the amount of available buffer storage after its execution.

ERASE

The ERASE function is the same as UNPLOT except that *all* instances of the display variables appearing as arguments are removed.

BLANK

The subroutine BLANK performs an ERASE on all active display variables, that is, it blanks the display device.

For example, if A had been plotted as in the above example, the statement

```
TX = UNPLOT(A)
```

would remove the current instance of A, that is, the characters generated by the PRINT. The statement

```
TX=UNPLOTA(A, A) or
TX=UNPLOTA(A) + UNPLOT(A)
```

would remove two instances of A, and

```
TX = ERASE(A)
```

would remove all instances of A.

READING ALPHAMERIC INPUT

At this point, we shall assume that the display device being used is an IBM 2250 with a buffer. In order to type alphameric input into a 2250 buffer, a cursor must be set in the buffer. GRAF provides a means of setting such a cursor in a display variable. When the display variable is plotted the cursor will be set in the buffer and be displayed on the screen. The characters which are typed on the keyboard replace the characters that were in the buffer and were being displayed. The new characters are then displayed on the screen.

READ

The READ statement is provided to read the contents of the display buffer into main storage. Its form is

READ (dv, format-number) list

If the display variable dv has an instance displayed on the screen, then all the character information from the *current* instance is read from the buffer into the display variable in main storage, converted according to the format specified, and sent to the variables in the list. If no instance was on the screen, the buffer is not read but the conversion is carried out with the contents of the display variable as it is.

Setting and removing cursors

A cursor can be set in a display variable either by a PRINT statement using the correct control character in its format or by the SETCUR subroutine. A cursor in a display variable is removed by the subroutine RMVCUR or by assigning a new value to the display variable by an assignment statement or by calling RESET.

There can be only one cursor on the screen at one time. The cursor is removed from the buffer if it is in an area of the buffer corresponding to an instance of a variable which has been removed by UNPLOT or ERASE. Also, a call to BLANK will remove the cursor.

ATTENTION HANDLING

In provisions for attention handling, GRAF shows most clearly the conditions under which the system was designed and developed. GRAF was to be used with Operating System 360 FORTRAN (E level) and to be implemented using Operating System 360 Express Graphic Support. The Express Graphic Support does not inform the user's FORTRAN program that an attention has occurred unless the FORTRAN program explicitly requests such information. Therefore, GRAF does not provide an elaborate method of handling attentions.

DETECT

The DETECT FORTRAN function returns information about the occurrence of attentions generated by the light pen, programmed function keyboard, end-of-order sequence, alphameric end key, and alphameric cancel key. The value of the function DETECT indicates the type of attention. Its form is:

DETECT (inqarray)

where inqarray is an array of dimension 5. DETECT returns information as follows:

```
if  NDET is an array of dimension 5 and
    J = DETECT (NDET)
```

were executed, the possible results are shown in the following table:

Table I. Results of an execution of detect

Value	*Meaning*
J = 0	no attention had occurred,
J = -1	a light pen attention occurred and |NDET (1)|=a number corresponding to the display variable causing the detect and then NDET (1) > 0 implies the current instance was detected or NDET (1) < 0 implies the current instance was not detected NDET (2) = x coordinate of detection point, in user coordinates NDET (3) = y coordinate of detection point, in user coordinates
J=1	a programmed function key attention occurred and NDET (4) = key number NDET (5) = overlay number
J=2, 3, or 4	an attention occurred corresponding to the end key, cancel key, or end-of-order sequence respectively and nothing is returned to NDET.

DETAIN

DETAIN works exactly the same as DETECT except that it waits until an attention occurs before returning, hence it never returns the value zero. DETAIN also allows the system to execute other tasks while the user program is waiting for a detect.

LPNAME

The value of the name of the display variable, the first element of the array inqarray (NDET (1) in the above example), is of little use without the LPNAME function which analyzes its meaning. The form of LPNAME is:

LPNAME(dvname, dc_1, dv_2,. . . ,dv_n)

The value of LPNAME is 0, 1, 2, . . . or, n. In using LPNAME, the variable dvname has a numerical value corresponding to some display variable, that is, a value returned to the first element of the array argument of a DETECT or DETAIN after a light pen detect. LPNAME determines whether or not that numerical value corresponds to any of the display variables dv_1, dv_2, ... ,dv_n. If a correspondence is found with say dv_j when the value returned by LPNAME is j. If no correspondence is found, then the value of LPNAME is zero.

The arguments of LPNAME can be dimensioned display variables. In that case, if dvname matches any element of the array the value of LPNAME will show a match with that array. In addition, if dv is an array, if the notation $dv/i_1/i_2/i_3$ is used, and if dvname matches an element of the three dimension array dv, then the subscripts where the match occurred will be returned into the integer variables i_1, i_2, i_3. Similarly one can use the notation with two and one dimensional arrays.

For example, if the following statements occurred in a program:

```
DIMENSION INQ (5)
DISPLAY S, T, U(5, 4), W(7, 3, 5)
    .
    .
    .
IB = PLOT (S, T, U, W)
    .
    .
    .
J = DETAIN (INQ)
```

and a light pen detect occurred, then

if T caused the detect

LPNAME (INQ(1), S, T, U, W) = 2, or

if U (2, 3) caused the detect

LPNAME (INQ(1), S, T, U, W) = 3

LPNAME (INQ(1), S, T, U/I/J, W) = 3 and I = 2 and J = 3.

The use of the name return from DETECT and DETAIN relieves the programmer of keeping track of his display by a separate bookkeeping scheme such as building a table of internal and external names.

In general, all the things which need to be plotted, erased, or detected with the light pen as distinct entities will be defined using distinct display variables or distinct elements of an array of display variables.

Miscellaneous attention functions

Other functions are provided to enable and disable attentions, and to turn the indicator lights on the programmed function keyboard on and off.

USER WRITTEN DISPLAY FUNCTION

The built-in display functions are PLACE, POINT, LINE, PRINT, and CHAR. The FORTRAN programmer can use GRAF to create display functions of his own which can be used in display assignment statements. To do this, he compiles a program in a way similar to compiling an ordinary FORTRAN function program. As the first statement of such a display function program, the statement DISPLAY FUNCTION is used. Its form is:

DISPLAY FUNCTION functionname (arg1, arg2, . . . , argn)

The rest of the program is written like a FORTRAN function program, but including GRAF statements.

For example, the programmer could create and use a display function called BOX as follows:

```
DISPLAY FUNCTION BOX (X, Y, U, V)
BOX = PLACE (X, Y) + LINE (U, Y) + LINE
  (U, V)+LINE (X, Y)
RETURN
END
```

The programmer can now use the display function BOX in a manner similar to the built-in functions PLACE, POINT, and LINE. For example:

```
A = BOX (0, 0, 2000, 2000) + BOX (3000, 2000,
  4000, 4000).
```

SUMMARY

We have defined a new type of variable, a display variable, in GRAF and have tried to extend FORTRAN in a consistent fashion to deal with the display variables. We have tried to minimize the number of new statements a programmer will have to learn in order to use a display device and at the same time, define a system that will be powerful enough to enable him to use all the capabilities of the display device. Further, we feel that coding, debugging and simply understanding the logic of a program from its listing are all made much easier by avoiding CALL statements with long argument lists for frequently needed graphic routines.

ACKNOWLEDGMENT

The authors would like to acknowledge the many useful discussions they have had with Dr. D. G. Cantor and Robert B. Keller.

REFERENCES

1 IBM Operating System/360 FORTRAN IV (E Level Subset) Form C28-6513
2 IBM System/360 Component Description
3 IBM Operating System/360 Graphic Programming Services for

The MULTILANG on-line programing system

by R. L. WEXELBLAT
Bell Telephone Laboratories, Incorporated
Holmdel, New Jersey

and

H. A. FREEDMAN
R.C.A. Laboratories, David Sarnoff Research Center
Princeton, New Jersey

INTRODUCTION

This paper describes the organization of an on-line programing system mechanized as an experiment in problem-solving.[1] Although not primarily a conversational system, a limited amount of interplay is possible. Designed as a bootstrapped operation, the subroutines associated with the operating system of the central processor may be called directly by a user's program and combined into larger entities. The system is built upon an information retrieval file system developed at the University of Pennsylvania and known as MULTILIST[2] which simulates a large content-addressable store of programs and data.

The basic philosophy of the programing language is to permit simple calls to initiate execution of classes of related user supplied programs and to present these programs with classes of data from the file.

At the time of this writing, user programs must be written in assembly language. Work is underway toward adding such languages as ALGOL, FORTRAN, IPL V and LISP I to the system.

Operation of the system

The system consists of a large-scale digital computer with disk file, cathode-ray tube keyboard consoles (character only) and a small computer in between to assist with input, output and editing. Figure 1 illustrates the flow of information in the system.

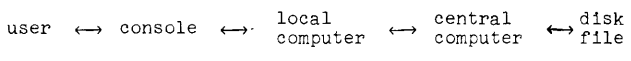


Figure 1 – Flow of information

The user types his data and commands to the system on the console keyboard. Limited editing facilities are available at the console, allowing the user to modify recently typed lines or characters by deleting and adding on the cathode-ray tube. Each complete page, up to ten lines, may be transmitted to the local computer which accumulates messages, edits and reformats them for the central computer. The central computer controls execution of commands, storage of information and programs on the disk file and the recall of programs and data from the disk. In a multiuser system, the local computer also has the duty of sequencing users and applying a priority control, interrupting the central computer whenever necessary. Results of computation return from the central computer to the local computer, thence to the console for display to the user.

A set of system programs controls operation of the system. These include an Executive Program for sequencing program execution, input and output, etc.; an Interpreter for executing queries and programs written in the Executive Language by the user; and a Storage and Retrieval System[3] for controlling a simulated associative memory in disk and core storage. The Storage and Retrieval System is based upon the MULTILIST technique for simulating an associative memory in a sequentially addressable memory. Its main advantage is the rapid, random access storage and retrieval of information to and from a disk file; a user of the system need not be concerned about the mechanics of storage and retrieval of data of programs.

The system and its Executive Language, MULTI-LANG, are designed for on-line use: a user sits before the console which has a cathode-ray tube for display, a keyboard for input and a teletype for hard copy; he types calls for processes and specifications of data directly into the computer which executes retrieved programs upon retrieved data and communicates the results to him.

Equipment configuration

The specific equipment configuration used to implement the pilot version of the system is illustrated in Figure 2. Bunker-Ramo Model 203 consoles are linked through a console control unit via DATA-PHONE to a PDP-5 computer. The PDP-5, acting as a local processor, performs the functions of editing input and output, reformatting input, scheduling and controlling priority for the various users on the system at any time. The local computer communicates through an adaptor with the data channel control of an IBM 7040 which acts as central processor. An IBM 1301 disk file is also linked with the 7040's data channel control.

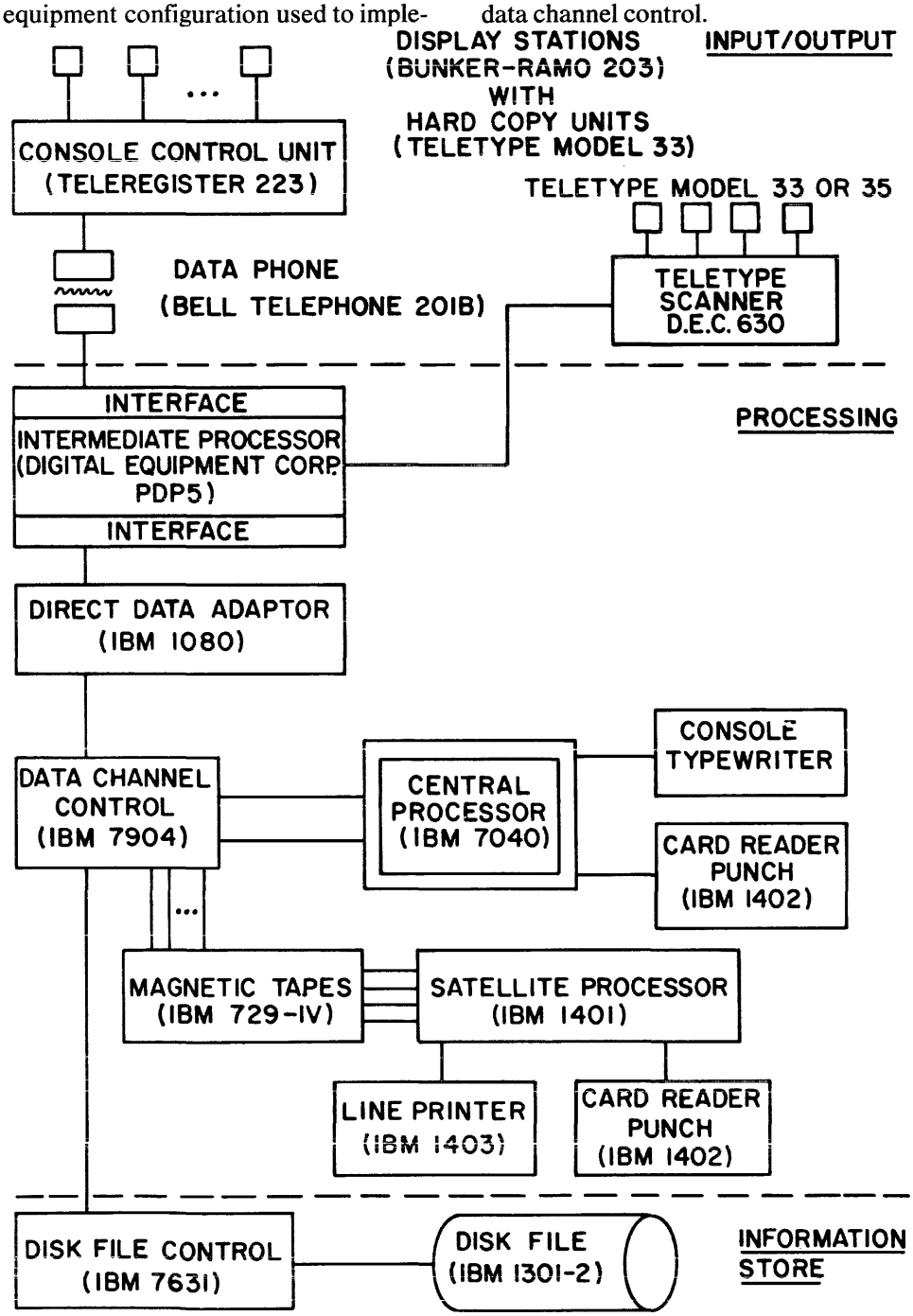


Figure 2 – The 7040/1301-PDP5 equipment configuration

The disk file serves as the main mass storage device for the system. The programs and data for every user, as well as many of the system programs, are kept on the disk. Information is kept on the disk until specifically deleted by a user. Other available equipment includes magnetic tapes, a console typewriter (output only), card reader/punch connected directly to the 7040, and an IBM 1401 computer with line printer and card reader/punch which can use tapes in common with the 7040. All of this equipment is available to the user through the system. Although the main processing path is Console to PDP-5 to 7040 to Disk File, the 1401 is useful for large scale input and output. The system is also programed to accept input from a card reader either directly or through the 1401 via magnetic tape and is able to give output to the line printer through the 1401. (Since the authors left the project, the hardware configuration has been extended to include model 33 and 35 teletypes as I/O devices.)

Although designed to run alone, the system has the ability to operate under control of the IBSYS operating system and, by suitable use of IBSYS control cards, may share the IBM 7040 with batch processing.

System organization

The main language of the system, MULTILANG, serves both as a control language and as a programing language, allowing use of the system on three levels:

- The inexperienced programer may make use only of programs already included in the system, although he may add new data as needed.
- The experienced programer may take fuller advantage of programs and data stored in the system by combining existing programs and data into procedures and adding new programs and data.
- The system programer may modify the system itself.

All information in the system is organized into "items," blocks of information made up of variable length records known as "elements." A user can refer to information by using a logical expression known as a "description" which specifies the desired data or program. The description, a basic part of MULTILANG, is composed of mnemonic, coded or natural language keywords; conditions upon the values of data in the item; and a listing of the specific parts of a multirecord item desired. Briefly, the programing language is made up of lines of coding called "statements" which may be grouped into programs called "procedures."

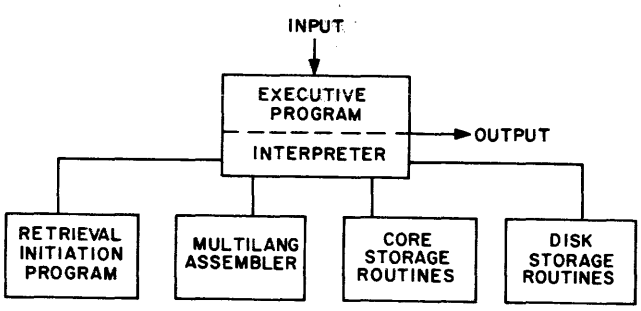


Figure 3 – System block diagram

A block diagram of the System Programs is given in Figure 3. The principal block consists of two parts: an Executive Program which controls input and output, conversion of data from external to internal format, assembly of programs and file loading; and a MULTILANG Interpreter which controls execution of MULTILANG programs and directs retrieval of programs and data.

The MULTILANG Assembler is an input conversion routine used to translate MULTILANG programs from external to internal format and assemble them into packed blocks.

The Disk Storage Routines control the sequencing and allocation of disk. The Core Storage Routines control the dynamic allocation of core storage for program and data.

The Retrieval Initiation Program is used by the Interpreter to initiate retrieval from disk and provide communication between the Interpreter and the disk memory. The Retrieval Initiation Program also serves as primary communication mechanism between the file and the user's program which can call it directly to obtain information. Retrieval may be according to descriptions supplied on-line by the user or according to descriptions generated internally by a program.

Information structure

The "item," the basic unit of information in the system, may contain program, data, or both. Items not currently in use are stored on disk and may be retrieved from disk to core for use either as programs or as data. Generally, the interpretation of a description in a statement initiates retrieval of programs to be executed. These programs may then request the interpretation of other descriptions and retrieval of items to be used as operands.

The general structure of an item is shown in Figure 4 to consist of a header, a set of keys, a linkword, a set of data elements and a table of contents. Each key is a 5-character string which serves to name or identify the information within an item. The elements may contain any type of information: data, program or both. Each element is identified in the item by an octal integer and the i^{th} key is said to be associated with the element bearing the number *i*. The header and linkword are used for internal bookkeeping. The table of contents serves to identify the elements with their numbers and to allow the system to locate elements in an item.

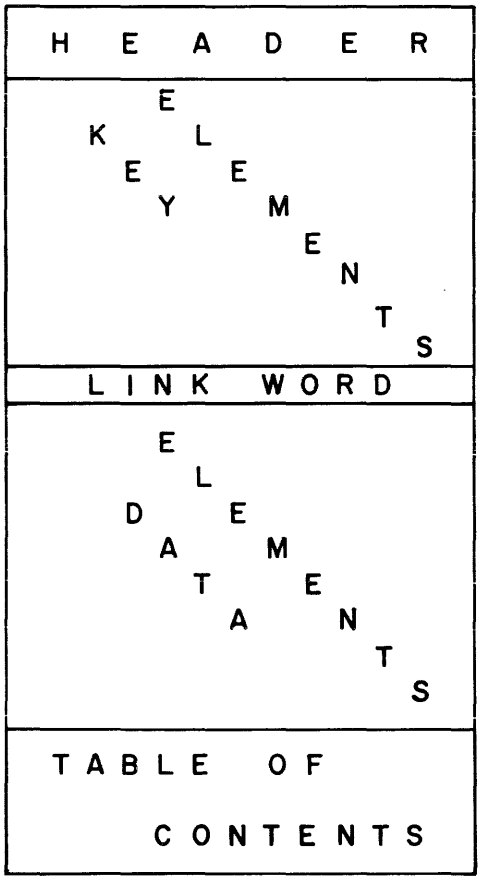


Figure 4 – Structure of an item

Descriptions

An item may be recognized or identified by the keys contained within it, the elements of information within it and the numeric value of the data within the elements of the item. In order to retrieve items, a "description," a logical expression combining keys, element identifying numbers and conditions upon the values of elements, is presented to the retrieval programs which then retrieve the desired items from disk storage, extract the requested information and place it in the high-speed memory. A formal specification of "description" is given in Table I.

Table I Formal Definition of "Description"

```
<description> ::= <primary part> <secondary part> <format part>

<primary part> ::= <key part> | (<key part list>)
<key part list> ::= <key part> v <key part> | <key part list> v
     <key part>
<key part> ::= <key> →<key> | <key value condition>
<key value condition> ::= <key> <relation> <value condition part>
<value condition part> ::= <key> | <element number> | <constant>

<secondary part> ::= <condition list> | <empty>
<condition list> ::=∧<condition> | <condition list>∧<condition>
<condition> ::= <simple condition part> |
     (<simple condition part list>)
<simple condition part list> ::= <simple condition part> v
     <simple condition part> | <simple condition list> v
     <simple condition part>
<simple condition part> ::= <existence condition> |
      <existence condition> | <value condition>
<existence condition> ::= <key> | <element number> |
     <key> →<key> | <element number> →<element number>
<value condition> ::= <key value condition> | <element number>
     <relation> <value condition part>
<relation> ::= < | <= | = | >= | > |  =

<format part> ::= <empty> | ,F(<element number list>)
<element number list> ::= <element number list>
     <element number> | <element number>
<empty> ::=
```

A description is made up of three parts as shown in Figure 5. The "primary" and "secondary" parts constitute a set of conditions which specify the items to be retrieved. A list of numbers, called a "format," specifies the elements of a retrieved item that are to be kept in core memory. At least one condition must be present in every description but the format may be omitted.

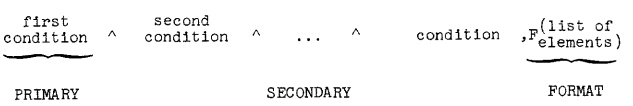


Figure 5 – Schematic representation of a description
(∧ stands for the logical "and")

Each condition may be an expression containing the logical connectives (inclusive) 'or' and 'not' so that the condition part forms a formula in Conjunctive Normal Form in which the terms are keys, element numbers or certain compounds containing keys and element numbers. The terms of this formula are known as "simple conditions" and the structure of the various legitimate simple conditions is indicated in Table II. Every simple condition is either true or false with respect to a given item.

Table II Sample Simple Conditions

1.	K	(K is a key, E is an element
2.	K	number, and C is a constant.)
3.	E > C	
4.	K > C	
5.	E1 < E2	
6.	E1 → E2	
7.	K1 → K2	
8.	(E1 → E2)	

In Table II and with respect to some given item, condition 1 is true if and only if the item contains the key. Condition 2 is true if and only if the item does not contain the key. Condition 3 is true when the value of the data contained in the element is numerically greater than the given constant. The fourth condition is true if the value of the data contained in the element associated with the given key is greater than the given constant. Condition 5 is true depending upon the values of the data associated with the two elements (and if the elements are present). Condition 6 is satisfied only if at least one element with number in the range E1, E1+1, E1+2, etc., up to E2 is present in the item. The seventh condition is similar to the sixth except that it concerns the value of the alphanumeric key (taken as a numeric bit string). The last condition in Table II illustrates the negation of a range condition.

To effect retrieval, a description is presented to the retrieval programs. The items retrieved will be those for which the formula in the condition part of the description is true. The retrieval scheme used in this system has been tested extensively using an encoded version of a bibliography on artificial intelligence.[4]

Table III illustrates a sample bibliography entry. Table IV shows correspondence between numeric element numbers and meaning of associated data. Table V gives sample descriptions.

Table III – The bibliography entry used in the example

Newell, A., and Simon, H. A., "The Logic Theory Machine," *IRE Trans. on Information Theory*, 1956 IT-2(3): 61-79.
This document is identified by the descriptors:
J_1 – Discussion of Heuristics for Machine Soltuion of Problems
J_2 – Discussion of Human Problem-Solving Heuristics
C_2 – Imperfect Searches Involving Failure, as Opposed to Decision Procedures
G_7 – Theorem Proving by Machine
G_8 – Use of Deductive Logic in Problem-Solving
I_2 – Grammatical Induction
H_4 – Symbol Manipulation Programing Systems
H_2 – Formal Languages
14 – Uses Special Programing System
34 – Report of Experiment
36 – General Discussion

Table IV – Contents of elements of sample item

Element	Contents
1	Date (year of publication)
4	Subject Codes
5	Authors' names (abbr. to 5 char.)
79,80	Authors' Full names
128	Full title, publisher, etc.

For the first description of Table V, every item in the file for which "SIMON" is a key would be retrieved; in this case, every document with Simon as an author.

The second description is somewhat more restrictive and will retrieve all documents written by Simon, published in 1956 and identified by key G7 (Theorem Proving by Machine). The format part specifies that only Element 128 (title, publisher, etc.) is to be kept.

Table V Some Sample Descriptions

	Numbers for Reference Only
SIMØN	(1)
SIMON ∧ 1956 ∧ G7, F(128)	(2)
NEWEL ∧ SIMØN ∧ ¬SHAW	(3)
(NEWEL ∨ SIMØN) ∧ H2 ¬H3, F(4,5,1,128)	(4)
32 ∧ (P4 ∨ I) ∧ A1 → A6	(5)
1956 ∧ E(5)==HOSIMØN	(6)

(=H Prefixes a 6 character constant)

The third description asks for documents by both Newell and Simon but not also by Shaw.

The fourth description specifies documents by Newell and/or Simon, containing key H2 (Formal Languages) but not Key H3 (Programing Language Systems).

The fifth description concerns purely subject codes, identifying documents that contain

32 (Extensive Bibliographies)

and either P4 (Self-Reproducing Machines)

$\vee$ $\llcorner SIM\phi n.$ could be added anywhere after the first term and before the format.) The format part states that elements 4, 5, 1 and 128 are to be kept.

The fifth description concerns purely subject codes, identifying documents that contain

32 (Extensive Bibliographies)

and either P4 (Self-Reproducing Machines)

or I (Inductive Inference Machines)

and also one or more of

A1 (Finite State Machines, Mathematical Theory)
A2 (Logical Network Theory)
A3 (Switching Theory)
A4 (Turing Machines)
A5 (Growing Machines)
A6 (Probabilistic Machines)

The preceding examples have accepted authors whether first or not. It is, however, possible to take advantage of the element value conditions (which consider only the first word of a multiword item) to seek first authors. The sixth description of Table V asks for all documents published in 1956 with (the first word of) element 5 equal to the constant OSIMϕN. Since element 5 is the author element, this is equivalent to specifying that the first author key be SIMϕN.

The language

A line of coding in MULTILANG is called a "statement." A statement may appear alone or with other statements in a MULTILANG procedure. Each statement corresponds roughly to a subroutine call with an arbitrary number of parameters. Upon interpretation, the MULTILANG Interpreter initiates the retrieval of the "described" routines and data. A schematic representation of a statement is given in Figure 6.

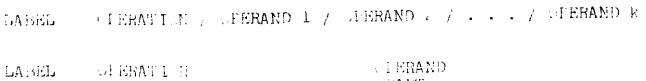


Figure 6 – The format of a statement

A statement may begin with a label-an octal integer of one to five digits - or it may have a blank first field. When present, the label serves to name or identify the statement so that the interpreter may call on it as a result of execution of another statement.

The operation part of the statement, which must always be present, is a description interpreted as describing the program to be executed. The Interpreter initiates retrieval using this description. Any retrieved items are considered programs. A description used for retrieval may identify no, one, or several items. If no items are found corresponding to the description of the operation, the statement is treated as a "no-operation" and is omitted. If the operation part description identifies one item and this item is a program item, the program will be retrieved and executed. If more than one program item is identified by the description, the programs are retrieved and executed one at a time in order of retrieval. Following the operation part of a statement are operands, each preceded by a solidus (/) and each may be any one of the following:

(a) a description,
(b) a local name (defined later),
(c) a label or element number, or
(d) a constant or block of constants.

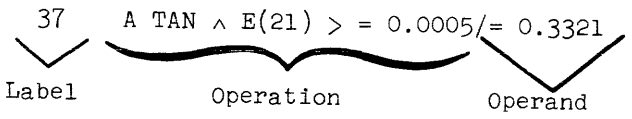


Figure 7 – Sample statement

Figure 7 illustrates a sample statement which will direct the Interpreter to retrieve and execute all programs having the key ATAN and an element numbered 21 with the value of element 21 greater than 0.0005. The decimal constant 0.3321 is the sole operand. In this case, assuming that element 21 contained some measure or precision, the statement could be calling for that routine with precision greater then 0.0005. It would be also possible to associate the precision figure with a key, say PREC and call for

ATAN ∧ PREC > 0.0005

The MULTILANG procedure

A "procedure" consists of one or more statements in MULTILANG together with certain identifying parts. An assembler converts MULTILANG procedures from the input format to a compact internal format, creating an item that may be stored in the file. A MULTILANG procedure can contain as many as seven parts. A formal specification of the syntax of "procedure" is given as Table VI.

(1) PROC (Procedure Name) Part:

Each procedure begins with the symbol "PROC" followed by the name of the procedure, a key of one to five characters which will be the first key of the item formed from the procedure

example: PROC SAMPL

(2) SYN (Synonym) Part:

Additional keys may be entered as alternate names for the procedure following the symbol "SYN" on the next line.

example: SYN STUFF,RLW, 234,5XBC

The keys STUFF, RLW, 234 or 5XBC are other names of the procedure by which it may be retrieved. This line is optional and may be omitted if no additional names are desired.

(3) *LOCNAM (Local Name) Part:*

Descriptions may be quite lengthy and it may be necessary for more than one statement in a procedure to use the same description. To avoid having to repeat the entire description, a "local name" may be defined to stand for, and be used in a statement in place of, the description. An arbitrary number of local names may be defined, each definition consisting of the symbol "LOCNAM" followed by the local name (a key), a solidus (/), then the description.

example: LOCNAM SEMI/QRP ∧
E(39) > E(97)

The local name "SEMI" is defined as equivalent to the description "QRP . . .".

A set of characters defined as a local name is so interpreted whenever it stands alone as an operand. Should the same set of characters be used as a key in a description, the second use will be independent of the first and no ambiguity will arise. Keys defined in PROC or SYN fields may also be used in descriptions or as local names.

(4) *BODY Part:*

The next section, present in every procedure, is the "body" consisting of one or more statements the first of which must be labelled. Normal execution of statements is first, second, third, etc., in succession, unless one of the subroutines executed specifies a "jump" to an out-of-sequence statement.

(5) *ENDSTA (End of Statements) Part:*

Following the last executable statement is the symbol "ENDSTA."

(6) *DATA Part:*

An arbitrary number of data elements may be assembled into a procedure item in the form of "pseudo-statements"—that is, sets of characters that look like statements but actually contain only data and are not executable. The format of a pseudo-statement is given in Figure 8. Each pseudo-statement generates a single data element and must contain at least one constant. Pseudo-statements must be labelled.

```
Label    Constant 1, Constant 2, . . . , Constant k
```

Figure 8 – Format of a data generating pseudo-statement

(7) *END Part:*

The end of a procedure is signalled by the symbol "END"; and if there are no pseudo-statements, the ENDSTAT may be omitted.

```
PROC     SAMPL

362      PROG

END
```

Figure 9 – A simple sample procedure

The minimum number of parts in a procedure is three: PROC followed by a key, one or more statements, and END. Figure 9 shows a simple procedure. This procedure, called SAMPL, has a single statement, labelled 362, which is a call upon the routine PROG.

Figure 10 illustrates a complete procedure. This procedure is called GAME with RENJU and MOKU as alternate names. VECTR is a local name corresponding to the description TERM ∧ E(21) > =123. The first statement is labelled "1" and calls for execution of all programs COMPV on the operand BOARD. There follow several other statements, some labelled. Following the ENDSTAT is one pseudo-statement labelled "7" which defines a data element containing the constants 300, 300, 200, 400, 0, 0 and 100.

Just as the example of Figure 9 tacitly assumed the existence of a program in the file known as PROG,

Table VI Formal Definition of "Procedure"

```
<procedure> ::= <proc part> <syn list> <locnam group>
     <statement group> <data group> END

<proc part> ::= PROC<procedure name>
<procedure name> ::= <key>

<syn list> ::= <empty> | SYN<synonym list>
<synonym list> ::= <synonym> | <synonym list> , <synonym>
<synonym> ::= <key>
<locnam group> ::= <empty> | <locnam list>
<locnam list> ::= <local name spec> | <locnam list>
     <local name spec>
<local name spec> ::= LOCNAM<local name>/<description>
<local name> ::= <key>

<statement group> ::= <labelled statement> <statement list>
<labelled statement> ::= <label> <unlabelled statement>
<label> ::= <element number>
<unlabelled statement> ::= <operation> | <operation>
     <operand list>
<operation> ::= <description> | <local name>
<operand list> ::= /<operand> | <operand list>/<operand>
<operand> ::= <description> | <local name> | E(<label>) |
     L(<label>) | <constant list>
<constant list> ::= <constant> | <constant list>,<constant>
<statement list> ::= <statement> | <statement list> <statement>
<statement> ::= <labelled statement> | <unlabelled statement>

<data group> ::= <empty> | ENDSTA <constant group>
<constant group> ::= <constant block> | <constant group>
     <constant block>
<constant block> ::= <label> <constant list>
```

the example of Figure 10 assumes the existence of programs names COMPV, MNMX, TEST, MATCH, OUTMV, INMV, STORE and QUIT.

In the statement labelled "1" the program (or programs) labelled COMPV will be retrieved and executed.

Upon request of the program, retrieval will be performed according to the description "BOARD" and the address, in core, of an item corresponding to the description (if such there be) will be given to the COMPV program. The program does not actually "know" what the description is; requesting retrieval according to its first operand, the actual

```
PROC      GAME
SYN       RENJU,MOKU
LOCNAM    VECTR/TERM ∧ E(21) > =123
 1        COMPV/BOARD
          MNMX/VECTR/BOARD/L(1)
 331      (TEST   MATCH)/STORE ∧ (E(23)==HWHITE
          E(23)==HOODRAW)/BOARD
          MOVE/L(1)/L(777)
          OUTMV
          INMV/L(1)/LEARN/BOARD/L(331)/E(7)
 777      STORE
          QUIT
ENDSTA
 7        =300,=300,=200,=400,=0,=0,=100
END
```

Figure 10 – A complete procedure

decoding of the description and its translation into retrieval requests is done by the system programs. In general, a worker program will request retrieval according to its *k*th operand and the system programs do the rest. If this operand is a description (or local name) retrieval is done. If the operand is a label or element number, the label or element number is given to the program. Should the operand consist of constants, the address of the first constant is given to the worker program.

A worker program has the option of informing the interpreter what to do next: continue to the next statement, terminate the procedure or skip to another statement. The statement above the one labelled 777 (which begins with INMV) illustrates these points. The first and fourth operands are labels, the second and third are descriptions. The last operand is an element number. An INMV program may request "retrieval" of operands one, two, ... or five. In each case, the result will be a code identifying the type of operand and the resultant data according to the rules given above. The program, upon completion, and depending upon the results of its computation may have the interpreter continue or go to statements 1 or 331. An attempt to execute any data element or nonexistent statement (undefined label) will cause termination of the procedure. Needless to say, the programs and data specified must be in the file before the procedure can be executed.

CONCLUSIONS

The combination of information retrieval scheme and interpreted control language described above was designed to be used with a group of machine-language subroutines or "worker programs" to form a basis for the problem-solving facility. At the time of this writing, the system had been used in two experimental applications: Bibliographic information retrieval and the processing of pictorial information.* A few examples of the former application are given below.

Although experimental in nature, the system has proved quite useful and has provided several interesting, albeit not revolutionary, conclusions:

- For on-line use, a cathode-ray tube console provides for much easier editing and quicker interaction than a typewriter only.
- For time-sharing on a medium-scale machine, an intermediate computer for processing input and output is useful.
- The ability to write and save programs in the command language is useful.

Examples of retrieval interaction

The following examples of use of the system were to demonstrate the various capabilities of the system. Five different subroutines were used:

1. *RETRE*—A worker program which retrieves from disk and prints all items corresponding to a description given as an operand.

2. *RETRX*—Similar to RETRE except leaves retrieved items in core storage and does not print them. A second operand specifies the key which will serve to identify the items in core.

3. *PRINT*—The second part of RETRE, to print items found in core. Any first operand is ignored, the second operand identifies the items in core to be printed.

4. *COUNT*—Counts the number of items satisfying a given description.

5. *ADAKY*—Retrieves according to a given description (first operand); adds to each item a new key in a specified position (second operand) and transfers to a label (third operand) if any items were modified. If no items are modified, transfer is made to another label (fourth operand).

*See the paper by A. van Dam to be given at this conference.[5]

```
XTRANS STAT

       RETRE/GORN,F(4)

XBEGIN
RESULTS OF RETRIEVAL

ITEM  1
    4:GORN          GORN, S.,

ITEM  2
    4:GORN          GORN, S.,

END OF RETRIEVAL



XTRANS PROC TESTB

11      COUNT/FOOF

        ADAKY/GORN/=HOOFOOF,=4,=0/L(11)/L(7)

        RETRE/GYR,F(3)

7       RETRE/GORN,F(4)

END

XBEGIN
ITEM COUNT   =    000
END ADAKY ... NUMBER OF ITEMS PROCESSED - 000002
              NUMBER OF ITEMS MODIFIED -  000002
ITEM COUNT   =    002
END ADAKY ... NUMBER OF ITEMS PROCESSED - 000002
              NUMBER OF ITEMS MODIFIED -  000000
RESULTS OF RETRIEVAL

ITEM  1
    4:GORN          GORN, S.,
      FOOF            --

ITEM  2
    4:GORN          GORN, S.,
      FOOF            --

XTRANS STAT

       ADAKY/PRINT/=HOIBBLE,=1,=0

XBEGIN
END ADAKY ... NUMBER OF ITEMS PROCESSED - 000001
              NUMBER OF ITEMS MODIFIED -  000001



XTRANS STAT

       RETRX'OR'PRINT/ALLAN,F(4)/=HOOPAUL

XBEGIN
RESULTS OF RETRIEVAL

ITEM  1
    4:ALLAN         ALLANSON, J. T.,

ITEM  2
    4:ALLAN         ALLANSON, J. T.,

END OF RETRIEVAL
```

```
XTRANS PROC TESTA

        RETRX/GORN,F(3,4)/=HOOOAAA

        PRINT/GORN,F(3,4)/=HOOOAAA

END

XENTER

XBEGIN
RESULTS OF RETRIEVAL

ITEM  1
     3:1959            --
     4:GORN          GORN, S.,
       FOOF            --

ITEM  2
     3:1957            --
     4:GORN          GORN, S.,
       FOOF            --

END OF RETRIEVAL
```

```
XTRANS STAT

        COUNT/GORN

XBEGIN
ITEM COUNT   =    002
```

```
XBEGIN TESTA
RESULTS OF RETRIEVAL

ITEM  1
     3:1959            --
     4:GORN          GORN, S.,
       FOOF            --

ITEM  2
     3:1957            --
     4:GORN          GORN, S.,
       FOOF            --

END OF RETRIEVAL
```

```
XTRANS STAT

      ADAKY/RETRX/=HOIBBLE,=1,=0

XBEGIN
END ADAKY ... NUMBER OF ITEMS PROCESSED - 000001
              NUMBER OF ITEMS MODIFIED -  000001
```

```
XTRANS PROC A

       COUNT/IBBLE

       IBBLE/ALLAN/=HOODICK

END

XBEGIN
ITEM COUNT   =    002
RESULTS OF RETRIEVAL

ITEM  1
    1:A0006           --
    2:I               --
    3:1956            --
    4:ALLAN         ALLANSON, J. T.,
    5:12              --
  104:ALLANSON, J. T.,
  200:SOME PROPERTIES OF A RANDOMLY CONNECTED NEURAL
      NETWORK, IN (I), CHAP. 30.

ITEM  2
    1:A0007           --
    2:F               --
    3:1956            --
    4:ALLAN         ALLANSON, J. T.,
    5:14              --
      11              --
  104:ALLANSON, J. T.,
  200:THE RELIABILITY OF NEURONS, IN (F).

END OF RETRIEVAL
```

Figure 11 – Examples from text

Table VII shows the sequence of inputs. Nine operations take place. Figure 11 is the teletype output of the run.

Example 1 – The fourth element of all items containing the key "GORN" is retrieved.

Example 2 – (To demonstrate looping) The number of items with key "FOOF" is counted. The key "FOOF" is added to each item with key "GORN."

The FOOFs are counted again if and only if a modification was performed.

Another attempt is made to add key FOOF (failing since the key is now present).

The GORNS are retrieved to show the new key.

Example 3 – Retrieval of more than one worker program by a given description was shown. The RETRX and the PRINT routines are applied in sequence to ALLAN.

Example 4 – The ADAKY routine adds the key IBBLE to the program PRINT. Since there are only two operands specified for the ADAKY program, no transfer will occur.

Example 5 – Procedure TESTA. RETRX, retrieves GORN specified by format elements 3 and 4, enters it into core under the key AAA.

The PRINT routine then displays them. In running the sample programs Procedure TESTS was entered into the disk file for later use.

Example 6—The instance of the COUNT program counting the number of GORNS.

Example 7—TESTA is retrieved from the disk and executed. The output this time will be the same as the output last time.

Example 8—The ADAKY program is used to add the key IBBLE to the worker program RETRX.

Example 9—Procedure A first counts the number of items containing the key IBBLE, two of them. Worker programs names IBBLE (meaning RETRX, and PRINT) are executed in sequence upon items containing the key ALLAN, no format specified.

The output as given by the computer is shown, consisting of a complete copy of the input and output. Three control words not previously discussed are shown.

XTRANS calls the translator (XTRANS STAT informs the translator that a single statement procedure follows).

XBEGIN begins execution.

XENTER saves the previous input for later use.

Table VII—Examples of retrieval requests in MULTILANG

```
  RETRE/GϕRN,F(4)                          1
  PRϕC TESTB
 11 CϕUNT/FOOF
  ADAKY/GϕRN/=HϕϕFϕϕF,
  =4,=O/L(11)/L(7)                         2
  RETRE/GYR,F(3)
  7 RETRE/GϕRN,F(4)
  END
  RETRX v PRINT/ALLAN,F(4)/
  =HOOPAUL                                 3
  ADAKY/PRINT/=HOIBBLE,
  =1,=0                                    4
  PRϕC TESTA                               5
  RETRX/GϕRN,F(3,4)/=HOOOAAA
  PRINT/GϕRN,F(3,4)/=HOOOAAA
  COUNT/GϕRN                               6
  TESTA                                    7
  ADAKY/RETRX/
  =HϕIBBLE,=1,=ϕ                           7
   PRϕC A
    CϕUNT/IBBLE                            9
    IBBLE/ALLAN/=HϕϕDICK
   END
```

REFERENCES

1 The work described in this paper was performed while the author was employed by the University of Pennsylvania on contract NOnr 551(48) (Operations Research Methodology Division, Office of Naval Research). The system described here is still under development and the description given is as of the fall of 1965. A more detailed description may be found in the reports on contracts NOnr 551(48) and NOnr 551(40), available through D.D.C.

2 N S PRYWES H J GRAY et al
The MULTILIST System
University of Pennsylvania The Moore School of Elect Eng Technical Report on contract NOnr 551(40), 2 vols November 1961

3 H A FREEDMAN
A Storage and Retrieval System for Real-Time Problem Solving
University of Pennsylvania The Moore School of Elect Eng Technical Report on contract NOnr 551(48), May 1965

4 An encoded version of a key word indexed bibliography M Minsky *A Selected Descriptor Indexed Bibliography to the Literature on Artificial Intelligence*
in E A Feigenbaum and J Feldman ed
Computers and Thought
McGraw Hill New York 1964

5 ANDRIES VAN DAM D EVANS
A Compact Data Structure for Storing Retrieving and Manipulating Line Drawings
Proc S J C C April 1967

RPL, A data reduction language

by FRANK C. BEQUAERT*
Computer Research Corporation
Newton, Massachusetts

INTRODUCTION

In the MITRE Interferometer Radar System, the outputs from a number of signal processors are recorded in digital form on magnetic tape utilizing an SDS-930 computer. A large number of these data tapes are recorded during system tests and observation of satellites. Data processing programs must be continually written to reduce these data to a form suitable for the analysis of radar performance.

In the past, the time taken from the specification of a data reduction program until an operational model was available was usually on the order of weeks for the simpler routines and months for more sophisticated programs. In addition, even with the use of FORTRAN, this programming involved a great deal of repetitive effort by the programmers.

With these problems in mind, in December 1965 development began on RPL (Radar Processing Language), a programming language to facilitate the writing of data reduction programs. The objective was a language that would produce useful data reduction programs from a typewriter conversation between a user and the computer. At this writing, a production model of the program is operational with the ability to generate relatively sophisticated data reduction programs.

Design objectives

The major objective in writing RPL was to produce a programming system that would automate as completely as possible all of the routine programming jobs normally associated with the MITRE radar data reduction operation. Specifically, it was desired that the program would be able to generate programs to perform the following functions:

1 Read data from records on a magnetic tape, unpack these data and store them in an organized manner in computer core.
2 Edit specified data values.
3 Generate simple functions of these data.
4 Print values of these functions in readable form.
5 Provide a method of plotting arrays of data.
6 Perform curve fits to arrays of data.
7 Perform correlation between sets of data.

Simplicity of operation was another design goal. It was desired that once the system was set up on the computer, an engineer with little knowledge of programming or machine operation could generate and execute useful data reduction programs through a typewriter conversation with the computer. It was hoped that the user would be able, in the course of an hour on the computer, to generate a program, test the generated program, and make one or more modifications (with subsequent tests) to the original program.

Finally, it was hoped that RPL would be directly useful to other facilities having recorded data to be processed.

Realization of objectives

Except for the correlation of data sets, all of the seven program generation features given above are currently operational in RPL.

The desired simplicity of operation was not fully realized. During the development of RPL it became obvious that the user of the system would require some knowledge of a computer language and of general concepts of computer programming. The specific areas of knowledge necessary to use RPL effectively are as follows:

1 A knowledge of the general concept of data arrays and of a notation for representing such arrays.

*Formerly with The MITRE Corporation, Bedford, Massachusetts.

"The work described in this paper was supported by the U. S. Air Force under Contract No. AF19(628)5165 monitored by the Electronic Systems Division of the Air Force Systems Command."

2 The knowledge of how to write FORTRAN-like arithmetic statements and the notations for such statements.
3 An understanding of the general concept of "flow" of operations in a computer program and the ability to generate simple program flow diagrams.
4 A knowledge of the difference between fixed and floating point numbers and their use.

Some acquaintance with the first three of these areas would seem requisite for programming in any language designed to manipulate data, no matter how "user orientated" it might be. For RPL on the SDS-930, knowledge of fixed and floating point notation is necessary to permit the user to conserve core storage in generated programs as fixed point arrays require half the storage space for the equivalent floating point arrays.

In general, standard FORTRAN notation is used for the representation of arrays and arithmetic operations as the generation of FORTRAN output statements from this input is extremely simple and most of the RPL users have had some FORTRAN programming experience.

Operation of RPL

RPL is a pre-compiler written in FORTRAN that generates FORTRAN output statements on magnetic tape. The program allows the use of a data base dictionary. This dictionary is used to specify the position and meaning of data recorded on a digital data tape. This dictionary is punched on cards. When a dictionary card deck is read by RPL, the program produces two outputs:

1 A dictionary listing on the line printer that gives data item names, dimensions, and descriptions (see Figure 1).
2 FORTRAN FUNCTION statements at the beginning of the RPL generated program. These statements stipulate, for the subsequentially generated program, the locations (word number and bit positions) within a data tape record where desired data may be found.

Once the dictionary deck has been generated for a particular class of data tapes, the user of RPL, no longer need concern himself with the position of data words within records on a data tape. The RPL dictionary system is capable of handling a wide variety of digital data tapes in which a particular piece of data on a tape is located by its position in a tape record. RPL will not currently handle tapes in which data is identified only by message labels.

RPL incorporates a number of program generation routines which generated sequences of FORTRAN output statements after typewriter conversation between the user and the computer. Currently the following types of program segments may be generated in this fashion:

1 *Input Variable Request (ASK).* A segment that requests a numerical input variable from the typewriter is generated in the output program.
2 *Data Tape Reading (GET).* A program segment is generated by GET that reads records of recorded data and generates arrays of functions of these data in computer core ready for printout, plotting, or further processing. Specified editing of data may be performed on the generated functions. The editing feature may be used either (1) to eliminate odd, spurious values from the generated data array or (2) to search for a specified value on a data tape.

The printout resulting from the use of the RPL GET function for a typical application is given below. The questions directed to the user and his *responses* are given at the left of the printout. The output FORTRAN statements generated by GET are preceded by asterisks.

```
 GET
ENTER TOTAL NO. OF ROWS OF DATA TO BE STORED
IN TABLE
 100
ENTER NO. OF FLOATING POINT ARRAYS TO BE MA-
NIPULATED BY GET
 1
ENTER FLOATING POINT ARRAY 1 NAME
 TIME
IS TIME SINGLY DIMENSIONED>
 YES
ENTER NO. OF FIXED POINT ARRAYS TO BE MANIPU-
LATED BY GET
 1
ENTER FIXED POINT ARRAY 1 NAME
 NUMBER
IS NUMBER SINGLY DIMENSIONED>
NO
ENTER SECOND DIMENSION SIZE
 5
THE ARRAYS AS NOW DIMENSIONED REQUIRE 700
LOCATIONS OF CORE.
IS THIS SIZE ACCEPTABLE>
 YES
   ********** CGET
   ********** 8000 DIMENSION TIME [  100]
   ********** 8001 DIMENSION NUMBER [  100, 5]
   ********** 8002 IQXCT = 0
   ********** 8003 CALL REED 5 [IWORD, 2925, IEOF,
                [N24,NPASS]
   ********** 8004 NQXROW = NUROW[IDUM]
   ********** 8006 DO 8005 NROW = 1,NQXROW
   ********** 8007 IQXCT = IQXCT + 1
ENTER EXPRESSION FOR TIME AS A FUNCTION OF
[NROW].
 ISEC [NROW] + IMIN [NROW]*60
   ********** 8008 TIME [IQXCT] = ISEC[NROW] +
                IMIN[NROW]*60
```

```
ENTER EXPRESSION FOR NUMBER AS A FUNCTION
OF [NROW, NCOL]
  IVAL [NROW, NCOL]
    ********** 8009 DO 8010 NCOL = 1, 5
    ********** 8010 NUMBER [IQXCT,NCOL] = IVAL
                [NROW, NCOL]
AFTER VARIABLES ARE EXTRACTED, THEY WILL BE
  EXAMINED FOR ANY SPECIFIED EDITING.
IF ANY VARIABLE IN A ROW FAILS THE EDITING.
  THE ENTIRE ROW WILL BE DISCARDED.
DO YOU WISH TO EDIT DATA>
  YES
ENTER ARRAY NAME ON WHICH YOU WISH TO EDIT
  TIME
TYPE LT IF YOU WANT TO DISCARD DATA LESS THAN
  A VALUE
TYPE EQ TO DISCARD DATA EQUAL TO A VALUE
TYPE GT TO DISCARD DATA GREATER THAN A VALUE
AND TYPE BET TO DISCARD DATA BETWEEN TWO
  VALUES
  EQ
ENTER VALUE YOU WISH TIME
        VALUE COMPARED WITH
    0 ********** 8012 IF [TIME [IQXCT]–0] 8013,8011, 8013

DO YOU WISH TO EDIT DATA>
  NO
WOULD YOU LIKE SENSE SWITCH TEST TO GIVE
OPTION OF EXITING BEFORE 100 ROWS ARE PRO-
CESSED>
  NO
              ********** 8013 IF [IQXCT – 100] 8005, 8014, 8014
              ********** 8011 IQXCT = IQXCT – 1
              ********** 8005 CONTINUE
              ********** 8015 GO TO 8003
              ********** 8014 CONTINUE
```

3 *Data Printout (PRINT).* The PRINT function may be used to generate a program segment to print arrays of data on the line printer. The user specifies any general table heading plus column headings he wishes printed, the names of the variables to be printed and the number of decimal places to be printed.

4 *Plotting of Data (PLOT).* Program segments to plot data arrays on the line printer may be generated by the RPL PLOT function. Two types of plots are available.
   1. A plot on a single page of line printer paper.
   2. A multiple page plot in which the independent variable runs along the edge of the printer paper and a single or double plot covers the width of the printer page.

5 *Curve Fitting to Data (FIT).* The FIT function in RPL generates output program segments that will perform least squares curve fits to arrays of data. The user specifies the names of the independent and dependent variable arrays on which the curve fit is to be performed, the number of points to be fitted from the arrays, the number of coefficients of the curve fit and the name of the coefficient array.

RPL is equipped with a number of features to permit the updating of old RPL generated programs and the modification of new program statements during program generation. The more significant of these features are as follows:

1 The user may copy any portions of a previously generated FORTRAN statement tape onto a new statement tape.
2 The user may backspace a statement tape under generation to the beginning of any specified statement.
3 Any errors in typewriter input that are detected before a line has been read into the computer may be corrected by hitting an error key and retyping the line.
4 Complete listings of either new or old FORTRAN statement tapes may be generated at any time.
5 At any time, b ytyping "EXIT" the user may exit from the middle of any program generation function back to general RPL control.

Features that allow additional flexibility of operation are as follows:

1 FORTRAN statements typed on the console typewriter may be written directly on the output statement tapes.
2 In all cases, punched card inputs may be substituted for typewriter inputs.
3 A complete record of all operations is kept on the line printer. This record gives computer generated typewriter output, users responses, and generated FORTRAN statements.

When the user has completed generation of a FORTRAN program with RPL, control is returned to the SDS MONARCH executive system. If the correct control cards are placed in the card reader, it is possible to compile, load and execute the generated program without user intervention.

RPL design

Initially, it was decided to write RPL as a pre-compiler written in FORTRAN which would generate FORTRAN output. This approach was taken because:

1 The SDS MONARCH executive system for the SDS-930 provided an automatic operating system into which a FORTRAN pre-compiler could be embedded.
2 The general form that FORTRAN data reduction programs should take was well known from previous programming experience.
3 Personnel with extensive FORTRAN programming experience were available.

4 Many of the programs generated by RPL would be kept as "production" programs for subsequent use. This requirement demanded that a relatively efficient and easily stored object program be produced. It was obvious that this objective could be realized by a pre-compiler.

The design of the RPL language was influenced by the available SDS-930 computer equipment. The MITRE SDS-930 system had neither a card punch nor a paper tape punch. As a result, all computer generated output that was to be subsequently used as computer input had to be written on magnetic tape. Thus, FORTRAN statements generated by RPL are written on magnetic tape and features are provided in RPL that allow for the reading of an old RPL generated FORTRAN statement tape and the generation of a new tape with deletion and insertion of FORTRAN statements. There is, however, nothing in the basic design of RPL to prevent a simple modification of the system to provide FORTRAN output on cards or paper tape.

The data base dictionary is read from a card deck by RPL at program initiation. The card deck consists of one record definition card followed by any number of data definition cards. The record definition card supplies the program with a description of the general makeup (record length, etc.) of the records to be read from the data tape. Each data definition card contains information (e.g., word and bit position) concerning one piece of data within the record.

As each data definition card is read by RPL a corresponding FORTRAN function is generated at the beginning of the output statement tape. A typical such function would be:

```
NAME(NROW)=IXTRAK(IWORD
(93+(NROW—1)*193),2,ϕ)
```

Here IXTRAK(NTABLE, NBITS, NPOS) is a machine language coded FORTRAN FUNCTION that extracts NBITS bits starting at bit position NPOS from core location NTABLE. IWORD is the array into which a tape record is read. NAME is the defined name of the data variable as specified on the data definition card. The numbers (93, 193, 2 and 0 in the above example) that specify the location of the data within the record (IWORD) read from the data tape are obtained from the record and data definition cards and inserted in the FUNCTION statement by RPL. With these FUNCTION statements at the beginning of the generated program, any reference to the specified variables (NAME in the above example) in the body of program will result in the extraction of the appropriate bits from the correct work of a data tape record. The variable names plus a definition (as specified on the data definition cards) are printed as a dictionary listing for reference by the RPL user (See Figure 1).

```
RPL DICTIONARY AS FOLLOWS FOR UTAPE AS SPECIFIED IN ROT DOCUMENT 11/29/65.
USE NROW TO INDICATE ROW IN TABLE.
USE IVAL TO INDICATE WHICH VALUE IN AN ARRAY OF IDENTICAL VALUES.

VARIABLE NAME                    DEFINITION
  NREC[NDUMMY]          UTAPE RECORD NUMBER
 NUROW[NDUMMY]          NUMBER OF ROWS OF DATA IN UTAPE RECORD
 ITTM6 [NROW]           TIME IN MICROSECONDS
 ITTM5 [NROW]           TIME IN 10S OF MICROSECONDS
 ITTM4 [NROW]           TIME IN 100S OF MICROSECONDS
 ITTM3 [NROW]           TIME IN MILLISECONDS
 ITTM2 [NROW]           TIME IN 100 TH OF A SEC.
 ITTM1 [NROW]           TIME IN 10TH OF A SEC. IN MICROTIME WORD
 ITTSEC [NROW]          TIME IN 10TH OF A SEC. IN MACROTIME WORD
IAZIMBC [NROW]          AZIMUTH OF BEDFORD ANTENNA (COMMAND) IN .044 DEGREES
IELEVBC [NROW]          ELEVATION BEDFORD ANTENNA (COMMAND) IN .044 DEGREES
ISDPRMH [NROW]          SDP RANGE FOR MILLSTONE IN MICROSECONDS
ISDPFBMH[NROW]          SDP FILTER BANK NUMBER FOR MILLSTONE
ISDPFNMH[NROW]          SDP FILTER NUMBER      FOR MILLSTONE
ISINIMH [NROW,IVAL]     SDP SIN OUTPUTS OF INTEGRATOR  MILLSTONE
ICOSIMH [NROW,IVAL]     SDP COS OUTPUTS OF INTEGRATOR  MILLSTONE
```

Figure 1—RPL dictionary listing

Programming of RPL

Programming of RPL and its program generation functions were found, in general, to be considerably easier than was initially expected. Approximately six-man-months of work by experienced FORTRAN programmers were required to write and debug the current RPL system. The FORTRAN program segments generated by RPL use library subroutines whenever possible for such operations as bit extraction, data plotting and curve fitting. Most of these subroutines had been previously developed and used extensively in hand coded FORTRAN data reduction programs. The use of these subroutines greatly simplifies the output program generated by RPL.

CONCLUSIONS

As of this writing RPL has been successfully used to generate a number of complete data reduction programs. In one case, a program consisting of 300 FORTRAN statements was generated and debugged in a single three hour period on the computer. It is estimated that the programming and debugging of this routine would have required at least two weeks of effort if hand coded in FORTRAN by a programmer experienced in the writing of data reduction programs.

In addition, RPL has been used to generate initial models of programs which were later greatly expanded by the addition of hand coded FORTRAN statements. This technique eliminates a great deal of the initial dogwork from the programming process.

A data reduction language of the form of RPL should be of value to a medium scale computer installation involved in the reduction of recorded data. It should be of particular value to an installation where the format of the recorded data and/or the required processing of data are subject to frequent change.

The size of the installation on which RPL will work effectively is probably critical. On a small system (i.e., less than 12K of core) it would become difficult to find a subset of RPL with reasonable enough flexibility. On a small machine the user would also be seriously limited as to the space available for generated programs. RPL is somewhat lavish with computer storage assigned by the generated program although the user is informed during program generation as to the size of any large blocks of storage assigned.

On a large scale computer, the cost of operation of the machine may prohibit sole use of the facility by a single program for hours at a time. In this case, it is possible to consider the use of some type of time-sharing. There should be relatively little difficulty in adapting a system like RPL to operate in a "real-time" environment in which a program under interrupt control (e.g., data recording) has priority over RPL operations, provided the necessary hardware and software (e.g., a real time monitor) are available for the machine in question. The operation of RPL within the framework of a more extensive time sharing system is another question. It is not clear for example, if the pre-compiler approach is the correct one for such an environment.

ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation to Mrs. Patricia Grassler and John Xenakis who participated in the design of the RPL system and wrote a major portion of the program.

An experimental automatic informational station AIST-O

by A. P. ERSHOV, G. I. KOZHUKHIN, G. P. MAKAROV, M. I. NECHERPURENKO and I. V. POTTOSIN

Computing Center of the Siberian Division of the USSR Academy of Sciences
Novosibirsk, USSR

INTRODUCTION

AIST-O is an experimental middle-scale time-sharing system. The name Automatic Informational Station (AIST) has been chosen to stress, by analogy, some new possibilities presented by time-sharing systems which give to a computation service some features of public informational and computational utility.

The development of the AIST-O station is being done in the Computing Center as a part of a more general activity known under the title "AIST project." The goal of this five-year program is (1) to provide the Computing Center by 1970 with adequate and modern centralized computing facilities which share their resources among many concurrently working users, (2) to find out what recommendations standard automatic informational stations and their software should have, and then transmit these recommendations to hardware manufactures and software designers. The "AIST project" activity has been stimulated to a great extent by the success of the time sharing development in the USA. One of the authors was acquainted with this development during his visit to this country in 1965. Some specific works in the field, namely: MAC Project CTSS,[1] Stanford Time-Sharing Project,[2] and RAND's JOSS system,[3] also influenced our approach.

The main task which has to be solved by the AIST-O construction is to obtain an initial experience in time-sharing system construction and the corresponding software development. At the same time it has to be a working system which will be in everyday use. Programming systems for AIST-O should cover to some extent all of the most interesting fields of applications of time-sharing. A special concern is to provide for the possibility of collecting various statistical characteristics of the system's functioning.

Hardware

From the engineering point of view it is not the goal to provide an optimal structure for the station. Our approach was defined, first of all, by the available equipment. Standard computers, Minsk-22, as a monitor, and two M-20 type computers, as working processors, are used. Recently a project on a multiprocessor computer system for batch processing has been elaborated in the Computing Center.[4] Many results of this project have been used in the AIST-O station for the control of the exchange with the

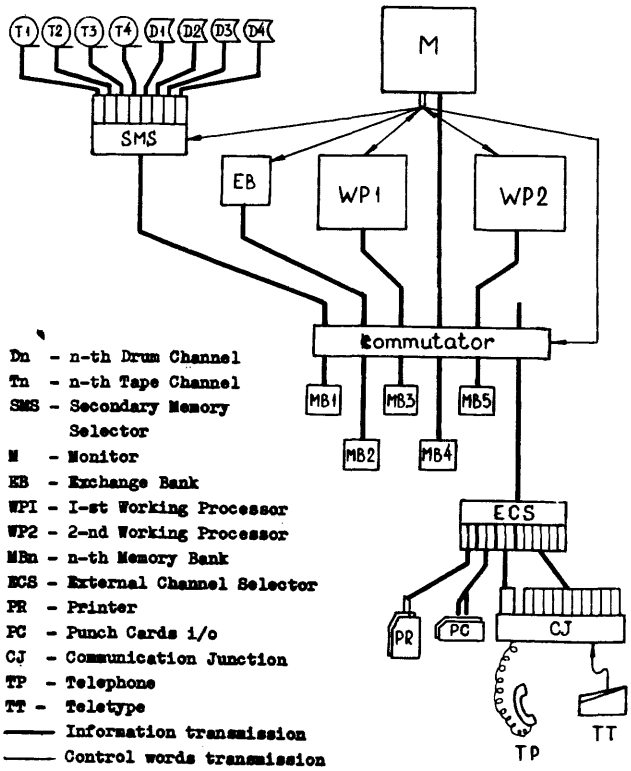


Figure 1—The AIST-O station structure

secondary memory, for internal channel switching and external channel switching. Figure I shows the AIST-O structure.

The commutator connects through information channels the active units of the station (processors, exchange bank, secondary memory selector) with the passive ones (memory banks, and external channel selector). Working frequency of the commutator is I megacycle, the speed of channel switching is about 100 μsec. The information channels have 45 information bits, 14 address bits and some number of bits for a reserve and parity check. The information channels between the working processors and the monitor, transfer the content of the control registers when the processors are interrupted or started. The information channels between the monitor and other units (commutator, secondary memory selector, exchange bank) are used to load control words into them and to start them. An active unit having been started, operates independently and after finishing its job sends a signal of its readiness to receive the next job. The external channel selector is treated in the same manner as the memory banks are from the point of view of information transmission. The "write into the buffer" signal is also a signal to transmit the information to the corresponding channel. When the buffer has been filled an interruption of the monitor occurs, but the actual transfer of the information from the buffer is performed under the monitor control.

The external channel selector (ECS) consists of a buffer memory, multiplexing block, and a control. The buffer memory is distributed among the following external channels:

10 telegraph channels with frequency of 60 bauds,
3 telephone channels with frequency of 600 bauds,
1 card reader channel with speed of 120 45-bit words per sec.,
2 card punch or line printer channels with speed of 60 words per sec. for printing and of 12 words per sec. for punching.

Every channel is connected with the interruption system and ends with an intermediate register.

The buffer memory is distributed among the channels in the following way.

Telegraph	-	8 words
Telephone	-	32 words
Card input	-	64 words
Print and card output	-	64 words

The telegraph and telephone channels are connected through a communication junction with a standard communication network.

The ECS control consists of the buffer control, channel number decoder, serial-to-parallel and vice versa transformers, the decoder which transforms control characters (carriage return, end of text) into interruption signals and buffer counters. The ECS interrupts the monitor when transfer from a buffer zone has been finished and when half a buffer has been filled (the channel buffer works as a swing: while one half of a buffer is receiving information from the channel, the other half transfers its contents to the internal channel, then both halves change their roles).

The ECS internal channel connecting the ECS and the commutator operates as a multiplexor under the guidance of the multiplexing block. The block contains an input register with one bit per channel. The external channel bits receive a "one" during every loading of the intermediate register of the channel. The internal channel bit receives a warning "one" 6 μsec. before an information transfer. The input register bits are tested in a revert-and cyclic order, i.e., the internal channel bit is tested after every testing of the next external channel bit in turn. The speed of a test is 1 μsec. per channel, the service time is 8 μsec. per 1-word portion of information from an external channel and 8 x p μsec. per p-words portion from the internal channel. Since $p \leq 32$ the service time for the high priority internal channel is no more than 300 μsec. which is much less than the speed of every external channel.

The *memory banks* are standard M-20 type memory modules (4K, 6 μsec. access time). Every commutator channel has up to 4 banks, so the general information capacity of the station is expendable up to 80 K words.

The exchange bank is introduced as a fast buffer and as a means for an exchange between memory banks. The exchange bank has its own control which permits it to operate independently on the monitor after receiving from it an exchange instruction.

The secondary memory selector (SMS) permits a concurrent exchange of up to 6 memory units (tape, drum) with any memory bank or ECS buffer. The SMS has a high-speed buffer (0.8 μsec. access time), 4 drum and 4 tape channels, a control and a multiplexing block for internal channel (between the SMS and the commutator). Every external (drum or tape) channel has 3 52-bit locations in the buffer. The first location is a buffer proper, the second one contains a control word, and the third stores a current location address in the memory bank involved in the exchange.

The multiplexing block contains an 8-bit register for a cyclic input of the external channels. One input of a channel takes 1 μsec. This time is enough to transfer the buffer content either into a secondary memory unit or the input register of a memory bank. Thus, the input of all the channels takes 8 μsec. This

time is small enough in comparison with the transfer speed through any of the external channels.

The working processors are central processors of the M-20 type computers with some modifications.

- an additional register for connection with the monitor,
- possibility of internal interruptions (traps and control transfer to the monitor),
- possibility of being started by the monitor.

The monitor is a Minsk-22 type computer with the following modifications.

- additional registers for communication with memory banks and other station units,
- an interruption "system" instructions (control word transfer, starting active units, communication with interruption system registers, transfer of 45-bit words into memory banks, connection with the clock).

The clock has a timer and several interval timers, i.e., reverse time-counters which, decreasing by one with every timer signal, send an interruption signal when reaching zero.

Software

General organization. All the program support is organized by a hierarchical principle. Elements of the hierarchical structure are called "system programs." For every i-th level system program there are several (i+I)-th level system programs available to it. This means that calls for some (i + I)-th level programs make sense and can be executed by i-th level program instructions. Every system program interacts with an external channel. The system may be in one of the three modes with respect to a text coming from the channel—a text execution mode, text processing mode, and text transmission mode. Furthermore, being in one of the first two modes, a program can understand or not understand a text. A program in a transmission mode simply passes a text without any analysis to an (i + I)-th level program which is connected with the corresponding external channel. If a program in a processing mode understands a text, then it can assimilate or analyze the coming text. Some special symbols in the text can switch the program to an execution mode. If a text contains an error (i.e., it becomes non-understandable), then the program sends a message to the external channel and turns into an execution mode.

A program in an execution mode tries to understand a text as an instruction for immediate execution. If the text is clear then the corresponding instruction is executed. If the text is not understood, then an (i − I)-th level program connected with the given one is switched to the execution mode and tries to execute the text. If the text is not understood even by a first-level program (dispatcher), then a message is sent to the external channel.

If an i-th level program A in an execution mode executes a call for an (i + I)-th level program B, then the program turns into the mode of a transmission of the text to the program B which turns into an execution mode.

The total swapping time for one memory bank is 400 msec. in AIST-O. So a special care is taken to maximally reduce the net swapping time. Following are the main aids to it:

- switching working processors among various memory banks,
- overlapping a swapping in one memory bank with a working usage of another bank,
- distribution of the stream of jobs among two processors for separation of long jobs from short ones,
- floating time quantum and scheduling based on an analysis of the job stream statistics (see below).

Some psychological measures will be taken to compensate a possible lack of reactivity. First, the station will be polite in the sense that if the time to execute a job is much more than the average waiting time, then the station will immediately send to the channel a calming request to wait a little. Second, some system programs will make the station slightly talkative. More wordy comments will reduce the frequency of inquiries and, moveover, will cause some thankful feeling of comfort because owing to detailed comments of the station a user himself can answer in a more laconic and convenient form.

The dispatcher is composed of first level system programs. The dispatcher programs are executed by the monitor.

From the organizational point of view the dispatcher is considered as a collection of subroutines serving as the primary reaction for interruptions plus several subroutines controlling other station units. The primary reaction is the minimal action necessary to save control registers and to identify a system program or a service subroutine responsible for the main actions which have to be initiated by the interruption.

Here is a list of the main service subroutines:

- The interpreter of system instructions and calls for the dispatcher. System instructions are those which are sent to the dispatcher from a console when the dispatcher is in an execution mode with respect to the corresponding channel. Calls for the dispatcher are those calls for its subroutines which come from system or users' programs.

- The physical exchange organizer. This subroutine receives data coming from other dispatcher subroutines and, using these data, forms control words, sends them to the commutator and the active station units and then starts them.
- The secretary. This subroutine keeps all operative records for users on line. Its main functions are identification of users, establishing correspondence between array and program names and their physical addresses, calculating the time and other operative accounting information.
- The scheduler. This program estimates quantitative parameters of inquiries of service, organizes a line of jobs and appoints the working processors for the service. A special feature of the dispatcher is a strict separation of the process of defining or specifying inquiry parameters from the process of line formation. This makes it possible to experiment with various schedules (see below).
- The editor. On the dispatcher level there will be only minimal editing of the text (character recoding, output line formation, printing of messages sent to the consoles by the dispatcher, etc.).
- The failure control. This subroutine periodically investigates station units and responds to interruptions caused by the information transmission check.

The system programs. The hierarchic structure of the program support makes it expandable so the list given below is incomplete and lists only those programs which are under construction now.

- The bookkeeper. This program keeps accounts of all the computational and informational service given to the users. Every user is considered by the station as himself as well as a member of a certain group, which is treated by the station as one whole from some point of view. For example, the summation of the computer time is done for every user individually but the time quota is given to all the group as a whole.

  The information kept by the bookkeeper includes means of identification of users and their groups and all the accumulated information. The bookkeeper works as a part of the dispatcher and as a separate system program making some accounting operations directly for a user.
- The file maintenance program. It is difficult to establish a comfortable file system without disk-file units. The users are supposed, however, to have a tape-oriented file system which will permit them to have individual files with possibilities to store, accumulate and change alphanumerical and binary information organized in lines, words, and pages. Files will usually be identified by their names. Physical addressing will be also available but in this case a user must keep in mind some constraints imposed by the station.

  There will be intermediate buffering files on the drum to reduce the interaction time.
- The batch processor. This program will be a supervisor for batched background programs. It is appropriate to mention that there exists no "user program" notion for the dispatcher. All the jobs, both background and foreground, are executed under the supervision of some system program.
- The console symbolic debugging system. In addition to standard characteristics of such systems we would like to mention one specific feature.

  It is well known that experienced programmers or console operators can greatly help a program author without any knowledge of the essence of a problem to be solved. This is because experienced programmers have a universal strategy of searching for bugs in programs. Such a programmer, following this strategy and combining it with a specific information taken from the author's answers given to the questions put by the programmer, leads the author in a way which permits him to find out the error.

  The problem is to try to discover this universal strategy and to implement it in a system program which could be called a consultant. The consultant will come to help by a user's request and, working in a conversational mode, will lead the user to success through a sequence of debugging operations.
- The incremental ALGOL compiler. The following goals stand for the incremental compiler: to make the compilation time imperceptible to a user, to inform him immediately about any syntactical and semantic errors that appear, to formulate and put questions to a user in such a form which permits the user to correct an error or to supply the compiler with a missing information in a more laconic and compact form than required by the ALGOL syntax.

  It is supposed that the syntax analysis, statement decomposition and a partial semantic analysis will be done during the conversation. The linking and final assembling in an absolute or relocatable form will be performed when the program has been composed and put into the station. The object program can be immediately executed or transferred to an individual file.

The incremental compiler itself will be a separate and shareable program interruptable at any moment.

- The analytic manipulation system. This system program is being developed by request of the mathematical departments at the Computing Center. Executive instructions in the system are requirements for various analytical manipulations (differentiation, integration with the help of integral tables, operations over polynomials, substitutions, simplifications, parenthesis expansion and so on). It will be a universal program working in a conversational mode when the general control is in the user's hands. The system, however, will be able to accumulate the analytical instructions for their later automatic execution in an interpretative mode of operation.

Scheduling and time slicing. A multiprocessor structure of the AIST-O station offers many possibilities for experimentation in selecting scheduling algorithms. A general approach to the scheduling algorithm analysis, simulation and implementation is briefly described below.

Let us describe an environment in which the scheduler S operates (for simplicity only one working processor P is considered). In the following, a "job" is understood to be a part of a real task which requires continuous work of the processor. This means that if a job J has been interrupted (because of either exchange operation or any other reason) then the job J is considered to be finished and the necessity to continue the job is then considered as a new job J^1.

The scheduler S contains a list Q of jobs $J_1, \ldots, J_n$ standing in a line. Every job J_1 has a set π_i of parameters which are necessary for the schedule composition. The processor P at the moment is running with a job J with parameters π. The job J has begun to run at the time t and has a time quantum Δ. Following are the external events for the scheduler S:

(a) Adding a new job J* to Q. A possible scheduler reaction is to interrupt job J, transform it into a job J^1 with parameters π^1. The job J* is sent to the processor P with a time quantum $\Delta^* = \Delta^*(\pi^*, Q)$. The job J^1 is added to Q.

(b) Rejection of a job from Q. A possible scheduler reaction may be an increase of Δ for the executed job J.

(c) Finishing the job J (stop or internal interruption). An obligatory scheduler reaction is sending a new job $\tilde{J}$ from Q and an appointment of a time quantum $\tilde{\Delta} = \tilde{\Delta}(\pi, Q)$ to it.

(d) Exhaustion of the time quantum Δ. An obligatory scheduler reaction is either an increase of Δ as in (b) or interrupt of J as in (a).

Let us consider in more detail a possible list of parameters of a job J. It has to be noted that some parameters characterize J not only as such but also as a part of some general task T. We consider the following characteristics to be useful.

- time t1 of entering the task T into the station,
- net processor time t2 spent to run the task T,
- number n of previous interrupts of the task T,
- supposed time t3 of completion of the task T,
- value V of the task T,
- time
- time t4 of entering job J in the list Q,
- supposed time t5 of completion of the job J,
- value C of the job J,
- time t6 for interruption and swapping of the job J.

A few words should be said of the sources and contents of these parameters. Some of the parameters (t1, t2, n, t4, t6) are directly defined by the dispatcher itself. The sources of the other parameters are system programs responsible for running the job J and the task Q. The sense of the parameters t3 and t5 is obvious and the only problem is to reliably predict them. The sense and quantification of V and C is much less clear a priori. Any inherent features of the tasks which may be important for better scheduling (for example, an absolute priority, the program length, etc.) can be related to V and C. Actually V and C can be a collection of scalar quantities.

The great degree of freedom in scheduling algorithms makes it difficult to select criteria for comparison or absolute evaluation of the algorithms. Comparison becomes more concrete if there exists a functional over a set of schedules. A degree of minimization of the functional value permits the estimation of the quality of the scheduler. Some possible alternatives for such functions are considered below.

(A) Cost functions. Let us suppose that at a moment in the list Q there appear at once n jobs $J_1, \ldots, J_n$. The solution time t_i and the cost C_i of storing J_i in the station during a time unit are known for every job. Then the total cost Φ of the solution of all the jobs is equal to

$$\Phi = \sum_{i=1}^{n} C_i T_i$$

where T_i is the time of finishing the job J_i.

It is known (for example[5] that Φ is minimized by such a schedule when all jobs are solved in turn and if they are ordered in Q by the values of the coefficient λ_i where $\lambda_i = t_i'/C_i$.

This algorithm is very attractive because of its simplicity but it has some serious deficiencies. The most obvious ones are difficulties in prediction of

times t_i and in an objective choice of the job values C_i. Besides this, as it has been shown by one of the authors, appearance of a new job J_{n+1} in the list Q during previous job running can destroy the optimum reached by the previous decisions about the jobs $J_i, \ldots, J_n$. It means that to save the t/C algorithm it is necessary to be able to predict the entrance of the new jobs. Sometimes it is really possible. For example, it is possible to make some predictions about the jobs which appear after finishing the exchange operations —such jobs are simply continuations of the old interrupted tasks.

(B) Preference principles. Sometimes a scheduler tactic is formulated as a principle which sounds like something similar to "shortest programs should be served first" or "the longer a job is in the station, the higher is its priority" and so on. A typical example of such an algorithm is a well-known "Corbato algorithm."[6] If A-type algorithms may be called "economical ones" then B-type algorithms may be called "political ones." These algorithms, without raising the problem of functional minimization, guarantee a minimum station sociability with respect to users.

(C) Combination of A- and B-type algorithms. The great number of degrees of freedom and the convenient computational form of the t/C algorithm permits the organization, within the frame of a cost function, pseudo minimization, of various scheduling algorithms. A variant of the algorithm will be formed by an appropriate choice of the values C_i and an appointment of the times t_i. History accumulation can be realized by an appropriate change of C_i as a function of the time. This approach has convenient, practical considerations because by properly choosing t and C it is possible to change the algorithm's tactics with respect to a class of jobs served by the station, without changing the algorithm itself or its tactics with respect to other jobs. The possibility of adaptation of the t/C algorithm to various scheduling tactics has also been mentioned.[5]

(D) Statistical scheduling algorithms. Let us suppose that the station deals with a stationary job stream with a known net service time distribution law. Then, if for a given time t there exists a positive probability to have jobs with the net service time t, it is possible to develop a mathematical expectation F(t) of the actual service time for these jobs. Naturally, $F(t) \leqslant$ for every t. Considering F(t) and t in some interval of t it is possible to introduce various functionals characterizing the "distance" between t and F(t). These functionals should be of an integral type, and they will differ from each other, for example, in time scale (logarithmic or linear scale) and in the weight function Φ (t) which multiplies the distance between t and F(t), for example, F(t)—t. A functional having been fixed, it is possible to compare and evaluate various scheduling algorithms. One of the possible formulations of the problem is: having an a priori given function F(t) one must try to find a scheduling algorithm satisfying this function.

The authors believe that this approach may be convenient for a scheduler evaluation based on real statistics of jobs coming to the station.

REFERENCES

1 *The compatible time-sharing system: A programmer's guide*
MIT Cambridge 1965

2 *Stanford time-sharing project STSP*
Memos 1-33 Stanford University Stanford 1963 1965

3 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
FJCC Proceedings 1964

4 *Some problems of multiprocessing in computing systems*
A Collection Nauka Novosibirsk in Russian 1965

5 M GREENBERGER
Priority problem
Project MAC report Cambridge 1966

6 I C PYLE
An outline of the MAC time-sharing system
STSP Memo 27 Stanford 1965

Some issues of representation in a general problem solver*

by GEORGE W. ERNST
Case Institute of Technology
Cleveland, Ohio

and

ALLEN NEWELL
Carnegie Institute of Technology
Pittsburgh, Penna.

INTRODUCTION

The research reported here is an investigation into the development of a computer program with general problem solving capabilities. This investigation involved the construction of one such computer program called the *G*eneral *P*roblem *S*olver (GPS, although more properly GPS-2-6) which was accomplished by modifying an existing program conceived in 1957 by A. Newell, J. C. Shaw, and H. A. Simon. (See references 1,2,3,4,5,6,7,8,9,10,11,12.)

The emphasis in this research is on the generality of GPS—on the variety of problems which GPS can attempt to solve. The quality of the problem solving exhibited by GPS is only a secondary consideration. Hence, the kind of problems for which GPS was designed are simple according to human standards. A typical problem is the missionaries and cannibals task in which there are three missionaries and three cannibals who want to cross a river. The only means of conveyance is a small boat with a capacity of two people, which all six know how to row. If, at any time, there are more cannibals than missionaries on either side of the river, those missionaries will be eaten by the cannibals. How can all six get across the river without any missionaries being eaten?

Another sample task is that of integrating, symbolically, a simple integral such as

$$\int te^{t^2} dt.$$

This problem is apparently quite different from the missionaries and cannibals task, but GPS has the generality, as well as the ability, to solve both of these problems.

Although GPS-2-5 was designed to be general, it, together with its predecessors, only solved three different kinds of problems due mainly to inadequate facilities for representing tasks. The central problem of this research is to generalize GPS-2-5 so that it can attempt a wider variety of problems. We also demand that the formulation of problems for GPS requires no knowledge of the internal structure of the program. Underlying this specific objective is the desire to shed light on some of the issues involved in designing better representations for problem solvers.

This is a brief statement of the problem. Section A gives a more detailed description of the problem on which this research focuses. The organization of GPS is described in Section B. Section C describes the representation of problems used by GPS-2-5 and Section D describes the generalizations incorporated in GPS. Section E gives a concrete example of a task solved by GPS. The results are summarized in section F.

A. The approach

We may consider a problem solver to be a process that takes a problem specification as input and, if

*This paper gives a brief summary of a more detailed work.[1] The extension of GPS described here is the essential part of the Ph.D. thesis of the senior author at Carnegie Institute of Technology.
We are indebted to H. A. Simon and J. C. Shaw for many discussions about the content of this work. This research was supported in part by Contract SD-146 from the Advance Research Projects Agency of the Department of Defense and in part by the Rand Corporation.

successful, provides the solution as output. Figure 1 provides a simplified picture. Initially the problem specification is expressed in an *external representation* which is converted by a translator to the *internal representation*—an encoding of the external representation inside the computer. The internal

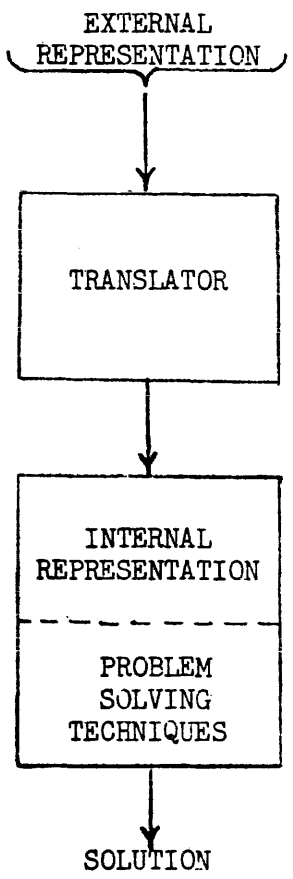


Figure 1 – A simplified model of a problem solver

representation is processed by a set of *problem solving techniques* and the result of this processing is (hopefully) the solution.

The model in Figure 1 is useful in depicting the issues on which this research focuses. For instance, this research ignores, by and large, the properties of good external representations; whereas the external representation is the central issue in some problem solving programs (e.g., the external representations of Bobrow[14] and Raphael[15] is an approximation to the vernacular). Rather, this research approaches the construction of a general problem solver by adopting a general paradigm of problems, heuristic search,[15] and then constructing problem solving techniques that are applicable to the paradigm. After describing heuristic search we will discuss the role of the internal representation of a general problem solver.

Heuristic search

In a simplified form of the heuristic search paradigm, there are objects and operators, such that an operator can be applied to an object to produce either a new object or a signal that indicates inapplicability. A heuristic search problem is:

Given:

a. an initial situation represented as an object,
b. a desired situation represented as an object,
c. a set of operators.

Find: a sequence of operators that will transform the initial situation into the desired situation.

The first operator of the solution sequence is applied to the initial situation, the other operators are applied to the result of the application of the preceding operator, and the result of the application of the last operator in the sequence is the desired situation.

The operators are rules for generating objects and thus define a tree of objects. Each node of the tree represents an object, and each branch of a node represents the application of an operator to the object represented by the node. The node to which a branch leads represents the object produced by the application of the operator. A method for solving a heuristic search problem is searching the tree, defined by the initial situation and the operators, for a path to the desired situation.

For many problems we know of no obvious heuristic search formulation. Thus, in some sense adopting heuristic search limits the generality that can be achieved. However, heuristic search derives its appeal from its generality, demonstrated by its wide use in other research efforts into problem solving (discussed in Chapter II of [1]).

The problem of generality

The power of a problem solver is indicated by the effectiveness of its problem solving techniques while its generality is indicated by the domain of problems that it can deal with.* The generality and the power of a problem solver are not independent because both depend strongly upon the internal representation. The internal representation is pulled in two directions: on the one hand, it must be general enough so that problems can be translated into it, and, on the

*This is an over-simplified statement since it qualifies a Turing Machine as a general problem solver. But its generality stems from the fact that the amount of information in the specification of a problem is not limited. For example, the problem of playing perfect chess can be given to a Turing Machine by listing all possible chess positions together with the best move for each position. But this specification, being impractical, does not qualify a Turing Machine as a chess player. We only point out the inportance of the amount of information in the specification of a problem; it will not be dealt with in this paper.

other hand it must be specific enough so that the problem solving techniques can be applied.

To illustrate this interdependence, consider a heuristic search problem solver whose only technique is to generate objects by applying the operators in a fixed order and testing if any of the generated objects are identical to the desired situation. It would be easy to construct such a problem solver with a relatively high degree of generality even though it could only solve the most elementary problems. On the other hand, it would be difficult today to achieve even a slight degree of generality with a problem solver that discovered the terms in an evaluation function for determining the likelihood of the existence of a path from any object to the desired situation. Thus, there are many different problems of generality, one for each set of problem solving techniques, and the difficulty of achieving generality depends upon the variety and complexity of the techniques.

This research investigates a particular problem of generality—the problem of extending the generality of GPS while holding its power at a fixed level. This involved extending the internal representation of GPS in such a way that its problem solving methods remain applicable and in a way that increases the domain of problems that can be translated into its internal representation. Thus, this research is mainly concerned with representational issues. We would not expect the issues to be the same in generalizing the internal representation of a problem solver which employed markedly different techniques than GPS. In this respect, this research has the nature of a case study.

B. GPS

GPS attempts problems by tree search, as does any heuristic search program. But to guide the search GPS employs a general technique called means-ends analysis which involves subdividing a problem into easier sub-problems. Means-ends analysis is accomplished by taking differences between what is given and what is wanted, e.g., between two objects or between an object and the class of objects to which an operator can be applied. A difference designates some feature of an object which is undesirable. GPS uses the difference to select a desirable operator—one which is relevant to reducing the difference. For example, in attempting the original problem, GPS detects a difference, if one exists, between the initial situation and the desired situation. Assuming that a desirable operator exists and that it can be applied to the initial situation, GPS applies it and produces a new object. GPS rephrases the original problem by replacing the initial situation with the new object and then recycles. The problem is solved when an object is generated that is identical to the desired situation.

The problem solving techniques of GPS consist of a set of methods, which are applied by a *problem solving executive*. To solve a problem, the problem solving executive selects a relevant method and applies it. Subproblems may be generated by the method in an attempt to simplify the problem. In such cases, the main problem may temporarily be abandoned by the problem solving executive for the purpose of solving the subproblem. Subproblems are attempted in the same way that the main problem is attempted—by selecting and applying a relevant method.

A complete description of the problem solving executive and the methods is given in Chapter III.[1] For the purposes of this paper, we will only illustrate the methods of GPS by describing how GPS approaches the missionaries and cannibals task. In this example INITIAL-OBJ* is the situation when 3 M (missionaries), 3 C (cannibals) and the BOAT are at the LEFT bank of the river. DESCRIBED-OBJ is the situation when 3 M, 3 C, and the BOAT are at the RIGHT. M-C-OPR, the only operator of this task, moves X missionaries, Y cannibals and the BOAT from the FROM-SIDE of the river to the TO-SIDE. We will ignore for the moment how the task, which includes the above objects and operator, is represented either externally or internally, assuming that there is some internal representation to which the problem solving methods of GPS can be applied.

Figure 2 shows the first few goals attempted by GPS. To solve TOP-GOAL, GPS matches the INITIAL-OBJ to the DESIRED-OBJ and detects that there are 3 too many cannibals at the LEFT. In attempting to alleviate this difference (GOAL 2) the M-C-OPR is applied with Y and FROM-SIDE specified to be 2 and LEFT, respectively. This operator application (GOAL 3) results in the OBJECT 1.

Since there are still too many cannibals at the LEFT, GPS attempts to move the remaining cannibal to the RIGHT (GOAL 4, GOAL 5, GOAL 6). However, first the BOAT must be moved back to the LEFT (GOAL 7, GOAL 8). The operator in GOAL 6 can be applied to OBJECT 2 which results in the old situation OBJECT 1. At this point GPS knows that it is in a loop and looks for something new to do. (To be continued in Section E.)

This example illustrates the kind of information that GPS must abstract from the internal representation in order to apply its problem solving

*We adopt the convention that words written in all capital letters correspond directly to IPL symbols inside the machine. These symbols are either defined in the IPL code[16] that comprises GPS or defined in the task specification.

TOP-GOAL: TRANSFORM the INITIAL-OBJ into the DESIRED-OBJ.
GOAL 2: REDUCE the number of C's at the LEFT bank of the river in the INITIAL-OBJ by 3.
GOAL 3: APPLY the M-C-OPR with Y = 2 and FROM-SIDE = LEFT, to INITIAL-OBJ. OBJECT 1: (LEFT (M 3 C 1) RIGHT (M O C 2 BOAT YES))
GOAL 4: TRANFORM OBJECT 1 into the INITIAL-OBJ.
GOAL 5: REDUCE the number of C's at the LEFT bank of the river in OBJECT 1 by 1.
GOAL 6: APPLY the M-C-OPR with Y = 1 and FROM-SIDE = LEFT, to OBJECT 1.
GOAL 7: REDUCE the difference that the BOAT is not at the LEFT bank in OBJECT 1.
GOAL 8: APPLY the M-C-OPR with TO-SIDE = LEFT to OBJECT 1.
OBJECT 2: (LEFT (M 3 C 2 BOAT YES) RIGHT (M O C 1))
GOAL 9: APPLY M-C-OPR with Y = 1 and FROM-SIDE = LEFT, to OBJECT 2.
OBJECT 1: (LEFT (M 3 C 1) RIGHT (M O C 2 BOAT YES))
GOAL 4: TRANSFORM OBJECT 1 into the INITIAL-OBJ.

Figure 2—The first few goals generated by GPS in solving missionaries and cannibals

methods. Hence, these methods place large demands on the internal representation because processes that abstract the information must be feasible. Below we summarize the demands of the problem solving methods, cross-referencing each to the above example.

Each of these demands requires that GPS employ a process for abstracting certain information from the internal representation. These processes may be different for different representations, but the information abstracted does not depend on the representation.

Object-comparison. GPS must be able to compare two objects to determine if they represent the same situation. *Object-comparison* is used in attempting the TOP-GOAL in Figure 2.

Object-difference. If two objects do not represent the same situation, GPS must be able to detect differences between them that summarize their dissimilarity. In attempting TOP-GOAL, the *object-difference* process detects the difference that is used in the statement of GOAL 2.

Operator-application. GPS must be able to apply an operator to an object. The result of this process is either an object, or a signal that the application is not feasible. The *operator-application* is used to achieve GOAL 3.

Operator-difference. If it is infeasible to apply an operator to an object, GPS must be able to produce differences that summarize why the application is infeasible. In attempting GOAL 6, the *operator-difference* process detects the difference used in the state of GOAL 7.

Desirability-selection. For any difference GPS must be able to select from all operators of a task those operators that are relevant to reducing the difference. (Of course, this selection will not in general be perfect.)

The M-C-OPR is selected to reduce the difference in GOAL 2. But before generating GOAL 3 GPS specifies the variables Y and FROM-SIDE to insure that the operator performs the desired function. Such a specification of variables limits the number of different ways that the operator can be applied to a given object. Hence, it can be viewed as the selection of a few promising possibilities from the total number of possibilities.

Feasibility-selection. For any object GPS must be able to select from all the operators those that are applicable to the object. (Again, perfect selection is not necessary.) This is meant to cover the case where the internal representation permits several operators of limited range to be combined into a single operator of wider range, such that the application of the unified operator does not decompose simply to the sequential application of the sub-operators. *Feasibility-selection* is used in achieving GOAL 3 in Figure 2. Note that the operator, move 0 missionaries and 2 cannibals from LEFT to RIGHT, is a schema in the sense that it can be applied to many different objects to yield many different results. For example, in GOAL 3 it is applied to INITIAL-OBJ to yield OBJECT 1 but it can also be applied to the object,

LEFT (M O C 2 BOAT YES) RIGHT (M 3 C 1),

to yield DESIRED-OBJ. *Feasibility-selection* requires that GPS produce the result without generating the different instances of the operator schema.

Canonization. GPS must be able to find the canonical name of certain types of data structures. Canonization arises from GPS's strategy for comparing two data structures. If they have canonical names, they are equivalent only if they have the same name. On the other hand, if two data structures do not have canonical names, they are equivalent only if all of their structure is equivalent. GOAL 9 in Figure 2 leads to the regeneration of GOAL 4. GPS recognizes that these two goals are indentical because they have the same canonical name (even though they are generated in different contexts).

C. Internal representation of GPS-2-5

The current version of GPS was developed through the modification of an existing version, called GPS-2-5. That GPS-2-5, together with its predecessors, solved only three different kinds of problems was due mainly to inadequate facilities for representing

tasks. The internal representation of GPS-2-5 will be described to clarify how the representation incorporated in GPS (described later) alleviated inadequacies in representation. The internal representation of a task for GPS (any version) consists of several different kinds of data structures:

a. objects
b. operators
c. differences
d. goals
e. TABLE-OF-CONNECTIONS
f. DIFF-ORDERING
g. details for matching objects
h. miscellaneous information

A complete description of the above types of data structures is given in Chapter IV.[1] Here, we will only describe the representation of objects and operators, which provide the main representational issues. However, an example of each kind of data structure is given in Section E. Other than objects and operators, differences are the only other type of data structure whose representation in GPS is different from its representation in GPS-2-5. Their representation depends to a large extent on the representation of objects and operators, and will be discussed in more detail later.

Objects. In GPS-2-5 objects are represented by tree structures encoded in IPL description lists. Each node of the tree structure can have an arbitrary number of branches leading from it to other nodes. In addition to branches, each node can have a local description given by an arbitrary number of attribute-value pairs. The tree structure in Figure 3, for example, represents the initial situation in the missionaries and cannibals task. In Figure 3 the node to which the LEFT branch leads represents the left bank of the river and the node to which the RIGHT branch leads represents the right bank of the river. The local description at the node which the LEFT branch leads to indicates that three missionaries, three cannibals, and the boat are at that bank of the river.

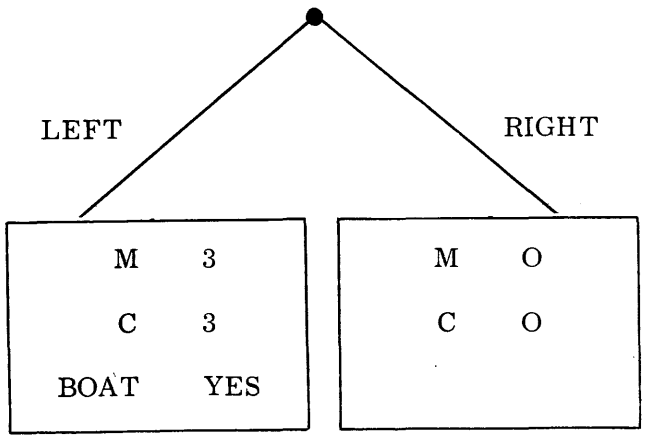


Figure 3—The tree structure representation of the initial situation in the missionaries and cannibal task

The use of variables in the tree structures described above allows a class of objects to be represented as a single data structure. For example, Figure 4 is the tree structure representation of $\int d^u du$. If u is a variable, this tree structure represents a large

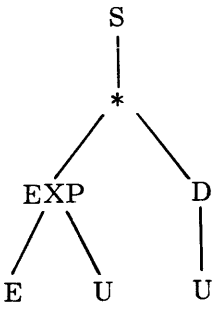


Figure 4—The tree structure representation of $\int e^u du$

class of objects. All members of the class have the same form but different values for u. GPS assumes that all tree structures may contain variables and it is prepared to process them as classes of objects.

Operators. In GPS-2-5 all operators were represented by representing the form of both the input and resultant objects. Assuming that u is a variable, Figure 4 is the tree structure representation of the input of the operator, $\int e^u du = e^u$, and Figure 5 is the tree structure of the output. Such an operator can only be applied to a member of the class of objects represented by the input form.

D. Representational issues

The representational issues that were investigated arose from various properties of tasks that could not adequately be dealt with by the existing program. Each of these issues will be discussed below. For some the representation was generalized, and the difficulty was removed. Other issues could not be dealt with within the framework of the existing program. However, attempting to alleviate these difficulties did clarify important aspects of the issues.

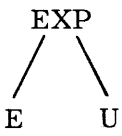


Figure 5—The tree structure representation of $\int e^u$

Desired situation

In many tasks the desired situation is a class of objects that could not be represented in GPS-2-5. In integration, for example, the desired situation is any expression that does not contain '$\int$'. A tree structure cannot represent this class of objects because all of the members do not have the same form. For this

reason the representation of the desired situation had to be generalized.

In introducing a new representation for the desired situation, GPS must be given some new processes for abstracting information from the new representation: a new *object-comparison* process so that GPS can compare an object to the desired situation; and a new *object-difference* process so that GPS can detect differences between an object and the desired situation. *Object-comparison* and *object-difference* are the only demands (described above) of GPS's problem solving method that are affected by the introduction of a new representation for desired situation.

The generalization of the desired situation allowed it to be represented as a set of constraints called a DESCRIBED-OBJ. A set of constraints represents a class of objects, each of which satisfies all of the constraints. The desired object in the integration task can be represented by the single constraint:

No symbol in the expression is an '$\int$'.

Each constraint in a DESCRIBED-OBJ is a data structure, called a TEST, that consists of a RELATION, and several arguments (in most cases, two). In the previous example,

a. the RELATION is NOT-EQUAL;
b. the first argument is a symbol;
c. the second argument is '$\int$'.

This constraint is quantified "for all" symbols. GPS recognizes NOT-EQUAL as a RELATION which it understands. (Currently GPS understands fifteen RELATIONS.) On the other hand, GPS only understands the generic form of the arguments, the arguments themselves being task dependent.

Using constraints to represent objects is convenient because each constraint is a simple data structure. Both the *object-comparison* process and the *object-difference* process analyze the structure of the constraints. The structure of many representations is too complex to permit such an analysis. For example, an alternate representation for the desired situation is a program whose input is an object and whose output is a signal indicating whether or not the object is a member of the class of objects that the program represents. The *object-difference* process for this representation would be extremely complex because it would require an analysis of the program.

Operators

The operators of many tasks, particularly mathematical calculi could be represented conveniently in GPS-2-5. However, the operators of other tasks could not, e.g., the operator of missionaries and cannibals. To alleviate this difficulty, in GPS an operator can be represented as a data structure, called a MOVE-OPERATOR, that consists of a group of TESTs and a group of TRANSFORMATIONs. The TESTs, which are the same as the TESTs in a DESCRIBED-OBJ, must be satisfied in order for the operator to be applicable and the TRANSFORMATIONs indicate how the resultant object differs from the input object.

A TRANSFORMATION is a data structure that consists of an OPERATION and several arguments. GPS knows the semantics of the OPERATIONs, but as in TESTs, only knows the generic form of the arguments, which are task dependent. Currently, GPS understands six OPERATIONs. A typical TRANSFORMATION (from the missionaries and cannibals operator that moves X missionaries, Y cannibals and the BOAT from LEFT to RIGHT) is:

> DECREASE the number of missionaries at the LEFT by X and increase the number of missionaries at the right by X.

In this TRANSFORMATION the OPERATION is DECREASE and the arguments are X, the number of missionaries at the LEFT, and the number of missionaries at the RIGHT.

Figure 6 illustrates how the operator that moves X missionaries and Y cannibals from LEFT to RIGHT can be represented as a MOVE-OPERATOR. The first two TESTs indicate that X and Y must be greater than O. The third TEST insures that at least one person is in the BOAT to operate it and that the capacity of the BOAT is not exceeded. The remaining TESTs prevent missionaries from being eaten.

TESTs:
1. $X \in \{0, 1, 2\}$
2. $Y \in \{0, 1, 2\}$
3. $X + Y \leq 2$
4. Either
   a. the number of missionaries at the LEFT $\geq$ the number of cannibals at the LEFT,
   or
   b. the number of missionaries at the LEFT = 0.
5. Either
   a. the number of missionaries at the RIGHT $\leq$ the number of cannibals at the RIGHT,
   or
   b. the number of missionaries at the RIGHT = 0.

TRANSFORMATIONs:
1. DECREASE the number of missionaries at the LEFT by X and increase the number of missionaries at the RIGHT by X.
2. DECREASE the number of cannibals at the LEFT by Y and increase the number of cannibals at the RIGHT by Y.
3. MOVE the BOAT from the LEFT to the RIGHT.

Figure 6—The MOVE-OPERATOR representation of the operator that moves X missionaries, Y cannibals and the BOAT from the LEFT to the RIGHT

The three TRANSFORMATIONs indicate how the application of the operator affects the number of missionaries, the number of cannibals and the BOAT, respectively. TRANSFORMATIONs can also implicitly test feasibility. For example, the BOAT must be at the LEFT in order for the third TRANSFORMATION to be applicable.

The introduction of MOVE-OPERATORs in GPS required the addition of new processes so that the problem solving methods could be applied to this new representation. New processes were needed for *operator-application, operator-difference, desirability-selection,* and *feasibility-selection.* Hence, the MOVE-OPERATOR representation was designed so as to make these processes simple. For many representations one or more of these processes would be too complex to implement.

A key feature of the MOVE-OPERATOR representation is its transparent structure. Each of the new processes does an analysis of this structure in order to abstract the necessary information. Another good property of MOVE-OPERATORs is its structural similarity to DESCRIBED-OBJ. This similarity causes the MOVE-OPERATOR processes to be similar to the DESCRIBED-OBJ processes, and thus all of these processes use the same basic subroutines. For example, the *operator-difference* process for MOVE-OPERATORs and the *object-difference* process for DESCRIBED-OBJs are nearly identical.

Unordered sets

The representation of some tasks requires representation of an unordered set. Multiplication, for example, can be represented as an n-ary function of a set of arguments whose order is unimportant. Such an unordered set can be represented in GPS-2-5 as an object, representing an ordered set, and an operator for permuting the elements of the set. This representation has the drawback that discovering the identity of two sets may require several applications of a permutation operator. The permutation operator would be unnecessary if the identity test could implicitly take into consideration the unordered property of the two sets.

The objects of GPS-2-5 can implicitly represent unordered sets, provided that the nodes can be tagged either ordered or unordered. These tags designate the branches of a node to be either ordered or unordered. Although a seemingly simple generalization, it considerably complicates the *object-comparison*, the *object-difference*, the *operator-application*, the *operator-difference*, and the *canonization* processes. These processes were generalized for the integration task so that the nodes of objects and operators could be unordered. Although the generalized processes were more complex and did more processing, there was a savings due to an overall reduction in the problem space.

The main complicating feature of unordered sets is that in matching two unordered sets corresponding elements must be paired. A variable can be made identical to any element via substitution and thus can be paired with any element. However, to see the identity of two unordered sets may require that a particular variable be paired with a particular element. Chapter V[1] discusses this issue in more detail and describes how the generalized processes (*object-comparison*, etc.) deal with this issue.

Large objects

GPS can only solve simple problems before its memory is exhausted. However, for some tasks the objects are so large that not even simple problems can be solved before its memory is exhausted. For example, the representation of a chessboard in GPS requires 1,000 memory locations and thus only several objects can be stored in memory.

There are two distinct difficulties with GPS's use of memory: (1) GPS saves in memory all objects generated during problem solving and (2) each object is a total situation, i.e., there is no provision for dealing with fragments of situations. These difficulties could not be dealt with in this research because they are too closely connected with the problem solving methods of GPS, which were held fixed.

Differences

In generalizing GPS, the representation of differences was degenerated. Each difference in GPS can only pertain to the value of an attribute of a node of an object. More global differences, such as the number of occurrences of a symbol, which could be represented in GPS-2-5, cannot be represented because they would introduce too much complexity in the *operator-difference*, the *object-difference*, and the *desirability-selection* processes. Thus, the generalization of these processes for MOVE-OPERATORs and DESCRIBED-OBJs was based on this simplified representation of differences.

Differences, although not part of the general heuristic search paradigm, are central to means-ends analysis, which is the main technique of GPS. Many tasks were not given to GPS, because the simple differences would not adequately guide GPS's search for a soltuion. For example, many of the logic tasks solved by GPS-2-5 cannot be solved by GPS due to the lack of direction provided by the degenerate differences. However, the representation of differences is adequate for the eleven tasks that were given to GPS.

E. An example

Since the emphasis of this research is on the internal representation of problems, it was designed without any consideration of how tasks might be communicated to the machine. The external representation was then designed so that a task expressed in it would be readable. Below we describe the external representation of the missionaries and cannibals task and how its corresponding internal representation is processed by GPS. It is hoped that the external representation of this task (and the other ten given to GPS[1]) is sufficiently readable for the reader to decipher. If this is the case, he can determine precisely what information is contained in the specification of a task. As noted earlier, the amount of information in the specification of a task is related to the issue of generality.

Figure 7 shows the specification of the missionaries and cannibals task for GPS. The specification of any task is a string of words delimited by "spaces." Parentheses are used to group the words. The first part of the specification (the part before the occurrence of TASK-STRUCTURES) indicates how words that are not part of GPS's basic vocabulary, should be interpreted. For example, BOAT is a word peculiar to this task. The string

BOAT = ATTRIBUTE

designates the BOAT to be an ATTRIBUTE of a node of an OBJECT-SCHEMA. This part of the task specification is analogous to declaration statements in ALGOL.

The remainder of the task specification defines the data structures that comprise the representation of a task. Each data structure definition has the form

<name> = (<body>)

The first such definition in Figure 7 follows TASK-STRUCTURES. The name of this structure is TOP-GOAL and it designates INITIAL-OBJ to be the "initial situation" of the problem and DESIRED-OBJ to be the "desired situation."

INITIAL-OBJ is the next data structure defined in Figure 7. This data structure, shown as a tree structure in Figure 3, represents the situation when the 3 missionaries, the 3 cannibals, and the BOAT are at the LEFT bank of the river. DESIRED-OBJ is a similar data structure representing the situation when the 3 missionaries, the 3 cannibals and the BOAT are at the RIGHT bank of the river.

The next four data structures are necessary only because they are used in the definition of M-C-OPR. X+Y is the name of a data structure representing the sum of X and Y. 1, 2 is the data structure representing the set that contains the two elements, 1 and 2. 0,1,2-SET and SIDE-SET are similar data structures that represent sets.

FROM-SIDE-TESTS is a data structure that represents a set of two TESTs. The first TEST is true if the number of missionaries at the FROM-SIDE is not less than the number of cannibals at that bank of the river. The second TEST is true if there are no missionaries at the FROM-SIDE. TO-SIDE-TESTS is a data structure similar to FROM-SIDE-TESTS. Both of these data structures are used in the definition of the next data structure, M-C-OPR.

M-C-OPR is the data structure that represents the only operator of this task. CREATION-OPERATOR indicates that the result of applying the M-C-OPR should be a new list structure instead of a modification of the list structure that represents the input object. The words enclosed in ('s are comments.

Following VAR-DOMAIN are four TESTs that constrain the legitimate values of variables. Incidentally, GPS knows which words are variables because they are listed in the data structure, LIST-OF-VAR, defined at the end of the task specification. The first two TESTs following VAR-DOMAIN require X and Y to be 0,1, or 2 and their sum to be no greater than 2.

The third and fourth TESTs indicate that both TO-SIDE and FROM-SIDE stand for different banks of the river.

The three TRANSFORMATIONs following MOVES indicate how the values of the three ATTRIBUTES, BOAT, M, and C, are modified in applying the operator.

```
RENAME

    LEFT = FIRST

    RIGHT = SECOND


DECLARE

    BOAT = ATTRIBUTE

    C = ATTRIBUTE
```

```
        B-L = FEATURE
        B-R = FEATURE
        C-L = FEATURE
        C-R = FEATURE
        DESIRED-OBJ = OBJECT-SCHEMA
        FROM-SIDE = LOC-PROG
        FROM-SIDE-TESTS = V-TESTS
        INITIAL-OBJ = OBJECT-SCHEMA
        M = ATTRIBUTE
        M-C-OPS = MOVE-OPERATOR
        M-L = FEATURE
        M-R = FEATURE
        SIDE-SET = SET
        TO-SIDE = LOC-PROG
        TO-SIDE-TESTS = V-TESTS
        X = CONSTANT
        X+Y = EXPRES
        Y = CONSTANT
        0,1,2-SET = SET
        1,2 = SET
    TASK-STRUCTURES
        TOP-GOAL = ( TRANSFORM THE INITIAL-OBJ INTO THE DESIRED-OBJ ; )
        INITIAL-OBJ = ( LEFT ( M 3 C 3 BOAT YES )
                        RIGHT ( M 0 C 0 ) )
        DESIRED-OBJ = ( LEFT ( M 0 C 0 )
                        RIGHT ( M 3 C 3 BOAT YES ) )
        X+Y = ( X + Y )
        1,2 = ( 1 2 )
        0,1,2-SET = ( 0 1 2 )
        SIDE-SET = ( LEFT RIGHT )
        FROM-SIDE-TESTS = (   1. THE M OF THE FROM-SIDE IS NOT-LESS-THAN
                                THE C OF THE FROM-SIDE ,
                              2. THE M OF THE FROM-SIDE EQUALS 0 ,
```

```
TO-SIDE-TESTS = ( 1. THE M OF THE TO-SIDE IS NOT-LESS-THAN
                     THE C OF THE TO-SIDE .
                  2. THE M OF THE TO-SIDE EQUALS 0 .
M-C-OPR = ( CREATION-OPERATOR
          $ MOVE X MISSIONARIES AND Y CANNIBALS FROM THE FROM-SIDE TO
            THE TO-SIDE $
          VAR-DOMAIN
          1. Y IS A CONSTRAINED-MEMBER OF THE 0,1,2-SET ,
             THE CONSTRAINT IS X+Y IS IN-THE-SET 1,2 .
          2. X IS A CONSTRAINED-MEMBER OF THE 0,1,2-SET ,
             THE CONSTRAINT IS X+Y IS IN-THE-SET 1,2 .
          3. THE FROM-SIDE IS AN EXCLUSIVE-MEMBER OF THE SIDE-SET .
          4. THE TO-SIDE IS AN EXCLUSIVE-MEMBER OF THE SIDE-SET .
        MOVES
          1. MOVE THE BOAT OF THE FROM-SIDE TO THE BOAT OF THE TO-SIDE ,
          2. DECREASE BY THE AMOUNT X THE M AT THE FROM-SIDE AND ADD
             IT TO THE M AT THE TO-SIDE ,
          3. DECREASE BY THE AMOUNT Y THE C AT THE FROM-SIDE AND ADD
             IT TO THE C AT THE TO-SIDE .
        POST-TESTS
          1. ARE ANY OF THE FROM-SIDE-TESTS TRUE ,
          2. ARE ANY OF THE TO-SIDE-TESTS TRUE . )
B-L = ( BOAT ON THE LEFT . )
B-R = ( BOAT ON THE RIGHT . )
C-L = ( C ON THE LEFT . )
C-R = ( C ON THE RIGHT . )
M-L = ( M ON THE LEFT . )
M-R = ( M ON THE RIGHT . )
DIFF-ORDERING = ( ( M-R M-L C-R C-L )
                  ( B-R B-L ) )
TABLE-OF-CONNECTIONS = ( ( COMMON-DIFFERENCE M-C-OPR ) )
COMPARE-OBJECTS = ( BASIC-MATCH )
BASIC-MATCH = ( COMP-FEAT-LIST ( M-L C-L B-L ) )
OBJ-ATTRIB = ( M C BOAT )
LIST-OF-VAR = ( FROM-SIDE TO-SIDE X Y )
END
```

Figure 7—The specification for GPS of the missionaries and cannibals task

Following POST-TESTS are two TESTs that must be true of any object produced by an application of the operator. These constraints prevent missionaries from being eaten in the resultant object. We can assume that the object to which the operator is applied satisfies these constraints because INITIAL-OBJ and DESIRED-OBJ satisfy them and all other objects are produced by an application of the M-C-OPR. The first TEST following POST-TESTS requires one of the two TESTs in FROM-SIDE-TESTs to be true (of the resultant object). This prevents missionaries from being eaten at the FROM-SIDE. The second TEST following POST-TESTS is similar. It should be noted that conjunction is the implied logical connective of the TESTs in M-C-OPR or any other MOVE-OPERATOR. And the TESTs in POST-TESTS of the M-C-OPR illustrate how disjunction can be used.

The next six data structures defined in Figure 7 are the types of differences of this task. A difference consists of a type of difference and a value. For example, the difference

C – L, -2

indicates that there are two more cannibals at the LEFT in one object than in another object. DIFF-ORDERING is the data structure that designates M-R, M-L, C-R and C-L to be more difficult to reduce than B-L and B-R.

TABLE-OF-CONNECTIONS indicates that M-C-OPR is relevant to reducing any type of difference. COMPARE-OBJECTS and BASIC-MATCH causes GPS to look for the following types of differences in comparing two objects:

a. M-L
b. C-L
c. B-L

Hence, GPS only compares the LEFT banks of two objects because what is not at the LEFT must be at the RIGHT.

OBJ-ATTRIB is a list of the ATTRIBUTEs and LIST-OF-VAR is a list of the variables.

Figure 8 shows the way that GPS attempts to solve the missionaries and cannibals task. In attempting TOP-GOAL, GPS detects that there are too many missionaries and cannibals at the LEFT. GOAL 2 is created in an attempt to reduce the number of cannibals at the LEFT by 3. GPS knows that there are 3 too many cannibals at the LEFT but the '3' does not get printed. GPS attempts to reduce the most difficult differences first which eliminates B-L. C-L was selected instead of M-L because GPS detected C-L first.

Since the TABLE-OF-CONNECTIONS indicates the M-C-OPR is relevant to reducing this difference, GPS attempts to apply it to INITIAL-OBJ. Before creating GOAL 3, GPS specifies Y and FROM-SIDE to be 2 and LEFT, respectively, so that the operator will perform the desired function. GOAL 3 results in OBJECT 5 and GPS attempts to transform this new object into DESIRED-OBJ (GOAL 4).

Since there are still too many cannibals at the LEFT (GOAL 5), GPS attempts to move the remaining cannibal to the RIGHT (GOAL 6). However, the operator cannot be applied because the BOAT is at the wrong bank of the river. In an attempt to bring the BOAT back to the LEFT (GOAL 7), GPS applies the M-C-OPR with the TO-SIDE equal to LEFT (GOAL 8) and OBJECT 6 is produced. GPS moves one cannibal across the river (GOAL 9); it does not realize that bringing the BOAT back to the LEFT also brought a cannibal with it. Since transforming the result of GOAL 9, which is an old object, into the DESIRED-OBJ is an old GOAL, GPS does not attempt it but looks for something else to do.

GOAL 11 is created in an attempt to transform all of the OBJECT-SCHEMAs which are derived from the INITIAL-OBJ, into the DESIRED-OBJ. (OBJECT 4, which is the SET of all OBJECT-SCHEMAs derived from the INITIAL-OBJ, is generated internally by GPS). GOAL 13 is created because OBJECT 6 has never appeared in a TRANSFORM type of GOAL. (This is the NEW-OBJ selection criterion.) Since there are too many cannibals at the LEFT in OBJECT 6 (GOAL 14), two are moved across the river (GOAL 15, OBJECT 7).

Everything goes smoothly until GOAL 27, which results in an old object, at which point GPS generates a GOAL indentical to GOAL 22. GPS does not reattempt GOAL 22 but generates a new GOAL (GOAL 29) by selecting a NEW-OBJ (OBJECT 8). Attempting GOAL 29 quickly leads to the old object, OBJECT 7 (GOAL 31) and the old GOAL, GOAL 16.

GOAL 33 is generated by selecting another NEW-OBJECT and GPS does not run into trouble until GOAL 49 which results in an old object. Again a new GOAL is generated by selecting a NEW-OBJ. GOAL 52 is abandoned because attempting it creates a GOAL identical to GOAL 43. The generation of GOAL 54 quickly leads to success.

F. Results

The initial motivation for this research comes from the tasks themselves; they could not be expressed in the internal representation of GPS-2-5. But the majority of this research focuses on the processing

```
1 TOP-GOAL TRANSFORM INITIAL-OBJ INTO DESIRED-OBJ      (SUBGOAL OF NONE)

  2 GOAL 2 REDUCE C-L ON INITIAL-OBJ      (SUBGOAL OF TOP-GOAL)

    3 GOAL 3 APPLY M-C-OPR WITH  Y = 2, FROM-SIDE = LEFT, TO INITIAL-OBJ      (SUBGOAL OF 2)
                SET:  X = 0, TO-SIDE = RIGHT
                OBJECT 5: (LEFT(M 3 C 1) RIGHT(M 0 C 2 BOAT YES))

  2 GOAL 4 TRANSFORM 5 INTO DESIRED-OBJ      (SUBGOAL OF TOP-GOAL)

    3 GOAL 5 REDUCE C-L ON 5      (SUBGOAL OF 4)

      GOAL 6 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 5      (SUBGOAL OF 5)
                SET:  X = 0, TO-SIDE = RIGHT

      5 GOAL 7 REDUCE B-L ON 5      (SUBGOAL OF 6)

        6 GOAL 8 APPLY M-C-OPR WITH  TO-SIDE = LEFT, TO 5      (SUBGOAL OF 7)
                SET:  Y = 1, X = 0, FROM-SIDE = RIGHT
                OBJECT 6: (LEFT(M 3 C 2 BOAT YES) RIGHT(M 0 C 1))

      5 GOAL 9 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 6      (SUBGOAL OF 6)
                SET:  X = 0, TO-SIDE = RIGHT
                OBJECT 5: (LEFT(M 3 C 1) RIGHT(M 0 C 2 BOAT YES))

  2 GOAL 11 TRANSFORM 4 INTO DESIRED-OBJ      (SUBGOAL OF TOP-GOAL)

    3 GOAL 12 SELECT FROM 4 A/C NEW-OBJ OF DESIRED-OBJ      (SUBGOAL OF 11)
                6 SELECTED

    3 GOAL 13 TRANSFORM 6 INTO DESIRED-OBJ      (SUBGOAL OF 11)

      4 GOAL 14 REDUCE C-L ON 6      (SUBGOAL OF 13)

        5 GOAL 15 APPLY M-C-OPR WITH  Y = 2, FROM-SIDE = LEFT, TO 6      (SUBGOAL OF 14)
                SET:  X = 0, TO-SIDE = RIGHT
                OBJECT 7: (LEFT(M 3 C 0) RIGHT(M 0 C 3 BOAT YES))

      4 GOAL 16 TRANSFORM 7 INTO DESIRED-OBJ      (SUBGOAL OF 13)

        5 GOAL 17 REDUCE M-L ON 7      (SUBGOAL OF 16)

          6 GOAL 18 APPLY M-C-OPR WITH  X = 2, FROM-SIDE = LEFT, TO 7      (SUBGOAL OF 17)
                SET:  Y = 0, TO-SIDE = RIGHT

          7 GOAL 19 REDUCE B-L ON 7      (SUBGOAL OF 18)

            8 GOAL 20 APPLY M-C-OPR WITH  TO-SIDE = LEFT, TO 7      (SUBGOAL OF 19)
                SET:  Y = 1, X = 0, FROM-SIDE = RIGHT
                OBJECT 8: (LEFT(M 3 C 1 BOAT YES) RIGHT(M 0 C 2))

          7 GOAL 21 APPLY M-C-OPR WITH  X = 2, FROM-SIDE = LEFT, TO 8      (SUBGOAL OF 18)
                SET:  Y = 0, TO-SIDE = RIGHT
                OBJECT 9: (LEFT(M 1 C 1) RIGHT(M 2 C 2 BOAT YES))

        5 GOAL 22 TRANSFORM 9 INTO DESIRED-OBJ      (SUBGOAL OF 16)

          6 GOAL 23 REDUCE C-L ON 9      (SUBGOAL OF 22)

            7 GOAL 24 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 9      (SUBGOAL OF 23)
                SET:  X = 0, TO-SIDE = RIGHT

              8 GOAL 25 REDUCE B-L ON 9      (SUBGOAL OF 24)
```

```
          9 GOAL 26 APPLY M-C-OPR WITH  TO-SIDE = LEFT, TO 9      (SUBGOAL OF 25)
              SET: Y = 1, X = 1, FROM-SIDE = RIGHT
              OBJECT 10: (LEFT(M 2 C 2 BOAT YES) RIGHT(M 1 C 1))

        8 GOAL 27 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 10     (SUBGOAL OF 24)
              SET: X = 1, TO-SIDE = RIGHT
              OBJECT 9: (LEFT(M 1 C 1) RIGHT(M 2 C 2 BOAT YES))

  3 GOAL 12 SELECT FROM 4 A/C NEW-OBJ OF DESIRED-OBJ      (SUBGOAL OF 11)
              10 SELECTED

  3 GOAL 29 TRANSFORM 8 INTO DESIRED-OBJ      (SUBGOAL OF 11)

    4 GOAL 30 REDUCE C-L ON 8      (SUBGOAL OF 29)

      5 GOAL 31 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 8      (SUBGOAL OF 30)
              SET: X = 0, TO-SIDE = RIGHT
              OBJECT 7: (LEFT(M 3 C 0) RIGHT(M 0 C 3 BOAT YES))

  3 GOAL 12 SELECT FROM 4 A/C NEW-OBJ OF DESIRED-OBJ      (SUBGOAL OF 11)
              10 SELECTED

  3 GOAL 33 TRANSFORM 10 INTO DESIRED-OBJ      (SUBGOAL OF 11)

    4 GOAL 34 REDUCE C-L ON 10      (SUBGOAL OF 33)

      5 GOAL 35 APPLY M-C-OPR WITH  Y = 2, FROM-SIDE = LEFT, TO 10      (SUBGOAL OF 34)
              SET: X = 2, TO-SIDE = RIGHT

    4 GOAL 36 REDUCE M-L ON 10      (SUBGOAL OF 33)

      5 GOAL 37 APPLY M-C-OPR WITH  X = 2, FROM-SIDE = LEFT, TO 10      (SUBGOAL OF 36)
              SET: Y = 0, TO-SIDE = RIGHT
              OBJECT 11: (LEFT(M 0 C 2) RIGHT(M 3 C 1 BOAT YES))

    4 GOAL 38 TRANSFORM 11 INTO DESIRED-OBJ      (SUBGOAL OF 33)

      5 GOAL 39 REDUCE C-L ON 11      (SUBGOAL OF 38)

        6 GOAL 40 APPLY M-C-OPR WITH  Y = 2, FROM-SIDE = LEFT, TO 11      (SUBGOAL OF 39)
              SET: X = 0, TO-SIDE = RIGHT

          7 GOAL 41 REDUCE B-L ON 11      (SUBGOAL OF 40)

            8 GOAL 42 APPLY M-C-OPR WITH  TO-SIDE = LEFT, TO 11      (SUBGOAL OF 41)
                  SET: Y = 1, X = 0, FROM-SIDE = RIGHT
                  OBJECT 12: (LEFT(M 0 C 3 BOAT YES) RIGHT(M 3 C 0))

          7 GOAL 43 APPLY M-C-OPR WITH  Y = 2, FROM-SIDE = LEFT, TO 12      (SUBGOAL OF 40)
              SET: X = 0, TO-SIDE = RIGHT
              OBJECT 13: (LEFT(M 0 C 1) RIGHT(M 3 C 2 BOAT YES))

      5 GOAL 44 TRANSFORM 13 INTO DESIRED-OBJ      (SUBGOAL OF 38)

        6 GOAL 45 REDUCE C-L ON 13      (SUBGOAL OF 44)

          7 GOAL 46 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 13      (SUBGOAL OF 45)
              SET: X = 0, TO-SIDE = RIGHT

            9 GOAL 47 REDUCE B-L ON 13      (SUBGOAL OF 46)

              9 GOAL 48 APPLY M-C-OPR WITH  TO-SIDE = LEFT, TO 13      (SUBGOAL OF 47)
                  SET: Y = 1, X = 0, FROM-SIDE = RIGHT
                  OBJECT 14: (LEFT(M 0 C 2 BOAT YES) RIGHT(M 3 C 1))

          8 GOAL 49 APPLY M-C-OPR WITH  Y = 1, FROM-SIDE = LEFT, TO 14      (SUBGOAL OF 46)
```

```
                    SET: X = 0, TO-SIDE = RIGHT
                    OBJECT 13: (LEFT(M 0 C 1) RIGHT(M 3 C 2 BOAT YES))

   3 GOAL 12 SELECT FROM 4 A/C NEW-OBJ OF DESIRED-OBJ      (SUBGOAL OF 11)
                    14 SELECTED

   3 GOAL 51 TRANSFORM 12 INTO DESIRED-OBJ      (SUBGOAL OF 11)

      4 GOAL 52 REDUCE C-L ON 12      (SUBGOAL OF 51)

   3 GOAL 12 SELECT FROM 4 A/C NEW-OBJ OF DESIRED-OBJ      (SUBGOAL OF 11)
                    14 SELECTED

   3 GOAL 54 TRANSFORM 14 INTO DESIRED-OBJ      (SUBGOAL OF 11)

      4 GOAL 55 REDUCE C-L ON 14      (SUBGOAL OF 54)

         5 GOAL 56 APPLY M-C-OPR WITH Y = 2, FROM-SIDE = LEFT, TO 14      (SUBGOAL OF 55)
                    SET: X = 0, TO-SIDE = RIGHT
                    OBJECT 15: (LEFT(M 0 C 0) RIGHT(M 3 C 3 BOAT YES))

      4 GOAL 57 TRANSFORM 15 INTO DESIRED-OBJ      (SUBGOAL OF 54)
SUCCESS
```

Figure 8—The performance of GPS on the missionaries and cannibals task

implication of several modes of representation. In generalizing GPS the trend appears to be toward richer representations whose structure is simple enough for GPS to "understand." For instance, GPS in some sense understands the M-C-OPR in Figure 7 because it can specify the pertinent variables so that the operator performs a desired function. On the other hand complex structures, such as IPL programs, were avoided in designing the internal representation of GPS.

The generalization of GPS has focussed on a particular group of tasks. If other tasks had been chosen, the generalization might have followed a quite different course. The tasks dealt with in this research were not chosen arbitrarily. Some categories of tasks, e.g., many optimization tasks, were deliberately avoided because we know of no obvious heuristic search formulation for them. Other categories of tasks were avoided because of deficiencies in GPS's problem solving methods. For example, games were avoided because GPS does not have a method for considering the opponent's moves and interests in addition to its own.

Of the tasks dealt with in this research, GPS can solve some, typified by the tasks described below. However, others cannot be solved by GPS due mainly to inadequacies in its representation. (See Chapter V[1] for a discussion of these inadequacies.)

We conclude this paper by briefly discussing the eleven tasks that were actually given to GPS. One of the instructive aspects of this research is the light shed upon the structure of these tasks. In addition, they serve as concrete examples of the level of generality achieved by GPS.

Missionaries and cannibals

GPS and one of its predecessors, GPS-2-2,[3] both solved the missionaries and cannibals task. The representation of the task in GPS was quite different from that used by GPS-2-2. The latter contains information about the nature of operators which the current GPS discovers for itself. GPS-2-2 was given ten operators: Move one missionary from left to right; move two missionaries from left to right: move one missionary and one cannibal from left to right, etc. The desirability of these operators for reducing the various types of differences was given to GPS-2-2, exogenously (in the TABLE-OF-CONNECTIONS). GPS is only given a single operator which moves X missionaries and Y cannibals across the river. In applying this operator GPS specifies the variables (X, Y, and the direction of the boat) so that the operator performs a desirable function.

GPS-2-2 was given a desirability filter for operators. This filter prevented GPS-2-2 from attempting to move more missionaries and cannibals across the river than there were on the side from which they were being moved. Such a separate filter is unnecessary in GPS because GPS never considers applying such an operator. Each operator in the GPS-2-2 formulation consisted of an IPL routine with its parameters (described on page 30[1]). The operator filter was also encoded in IPL. Not only is it tedious to construct IPL routines but the construction of these routines

requires some knowledge of the internal structure of GPS-2-2. The construction of the single operators given to GPS is much less tedious and requires no knowledge of the internal structure of GPS.

Integration

GPS symbolically integrated $\int te^{t^2}dt$ and $\int(\sin^2(ct)\cos(ct) + t^{-1})\,dt$.

In the integration, multiplication and addition are represented as n-ary functions whose arguments are represented as an unordered set. Thus, the commutativity and associativity of multiplication and addition are expressed implicitly instead of representing them explicitly as operators. If they were explicitly given to GPS as operators, there would be an overall increase in the problem space which would prevent GPS from solving some trivial integrals.

SAINT,[17] a program that is quite proficient at symbolic integration, also represents the commutativity and associativity of multiplication and addition implicitly. Other similarities and some dissimilarities between SAINT and GPS are discussed in Chapter V.[1]

Tower of Hanoi

In the Tower of Hanoi, which is a classical puzzle, there are three pegs and a number of disks, each of whose diameter is different from all of the others. Initially, all of the disks are stacked on the first peg in order of descending size. The problem is to discover a sequence of moves that will transfer all of the disks to the third peg. Each move consists of removing the top disk on any peg and placing it on top of the disks on another peg, but never placing a disk on top of one smaller than itself. GPS solved the 4-disk Tower of Hanoi task.

The Tower of Hanoi is an example of a task for which means-ends analysis is very effective. GPS never makes a mistake on this task, mainly because the differences and the DIFF-ORDERING are in some sense optimal. For many tasks, it is difficult to find good differences and a good DIFF-ORDERING. For a different treatment of this task see reference 18.

Proving theorem in the predicate calculus

GPS proved the following simple theorem in the first order predicate calculus:

$$(\exists u)\,(\exists y)\,(\forall z)\,((P(u,y) \supset (P(y,z) \,\&\, P(z,z))) \,\&$$

$$((P(u,y) \,\&\, Q(u,y)) \supset (Q(u,z) \,\&\, Q(z,z))))$$

$\exists$ is the existential quantifier; $\forall$ is the universal quantifier; P and Q are predicates; u, y and z are variables; $\supset$, &, and V are implication, conjunction and disjunction, respectively. The formulation of this problem is basically the same as that used by Robinson.[19]

Perhaps the most instructive part of this example is the light it cast upon the evolution of problem solving programs. In LT,[20] a theorem proving program for the propositional calculus, which is the predecessor of GPS, it was noted that the match routine was the source of most of the power of the program over a brute force search. GPS may be considered as an attempt to generalize the match routine, based on that experience. The first predicate calculus theorem prover did in fact use brute force search.[21] From an efficiency point of view the main effect of the resolution principle of Robinson was to reintroduce the possibility of matching (gaining, thereby, a vast increase in power). And it is this feature that allows GPS to use the resolution principle in a natural way.

Father and sons task

GPS solved the task in which a father and his two sons want to cross a river. The only means of conveyance is a small boat whose capacity is 200 pounds. Each son weighs 100 pounds while the father weighs 200 pounds. Assuming that the father and either son can operate the boat, how can they all reach the other side of the river? Of course, there is no way to cross the river except by boat.

This task is very similar to the missionaries and cannibals task. Both tasks involved moving two different kinds of people across a river in a small boat. But their formulations for GPS are quite different, in that none of the operators, objects, or differences are the same. Many of the earlier publications on GPS (e.g., [6]and[7]) make the distinction between a task and a task environment – the common part of a group of similar tasks. The father and sons task and the missionaries and cannibals task muddies this distinction. On the one hand, they should both have the same task environment because of their similarity. In fact they have different task environments because none of their objects, operators and differences are the same.

Monkey task

This task, which GPS solved, was invented by McCarthy[22] as a typical problem for the Advice Taker program.[23] In a room is a monkey, a box, and some bananas hanging from the ceiling. The monkey wants to eat the bananas, but he cannot reach them unless he is standing on the box when it is sitting under the bananas. How can the monkey get the bananas? The answer is that the monkey must move the box under the bananas and climb on the box before he can reach the bananas. The problem originates in the study of the problem solving ability of primates; its interest lies not in its difficulty, but in its being an example of a problem subject to common sense reasoning.

It is interesting to compare GPS's formulation of this task to the formulation for a typical Advice Taker program.[24] In GPS the objects are models of room configurations where as in 24 the objects are linguistic expressions that describe certain aspects of the room configurations. Both representations have advantages. For example, linguistic expressions are useful for representing imperfect information such as the monkey is in one of two places. However, models can represent implicit information that has to be represented explicitly when linguistic expressions are used, e.g., the monkey can only be in one place at a time.

Three coins puzzle

In this task there are three coins sitting on a table. Both the first and third coins show tails, while the second coin shows heads. The problem is to make all three coins the same—either heads or tails—in precisely three moves. Each move consists of "turning over" any two of the three coins. For example, if the first move consisted of turning over the first and third coins, all of the coins would be heads in the resulting situation. But the task is not solved because only one move was taken instead of the required three.

The peculiarity of the three coins task is in the solution being constrained to a fixed number of operator applications. This constraint was handled in GPS by expanding the representation of objects to include a counter that indicates the number of operator applications involved in producing the objects. In the desired situation the counter must have a particular value.

Parsing sentences

GPS parsed the sentence,

Free variables cause confusion. ,

according to a simple context-free grammar. (It contains ten productions or rewrite rules.) A great deal of effort has been devoted to the construction of efficient parsing algorithms for simple phase structure grammars. The point of this example is not GPS's proficiency as a parser, but to illustrate the kinship between heuristic search and syntatic analysis.

Bridges of Konigsberg

In the German town of Konigsberg ran the river Pregel. In the river were two islands connected with the mainland and with each other by seven bridges as shown in Figure 9. How can a person walk from some point in the town and return to the same point after crossing each of the seven bridges once and only once? In 1736 Euler proved that this task is impossible, and his proof stands as one of the early efforts in topology.

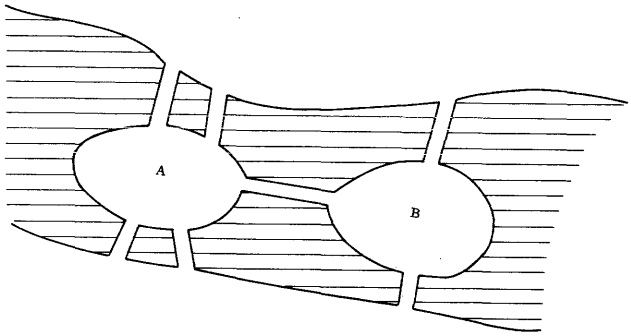


Figure 9—A schematic of the seven bridges of Konigsberg The shaded area is the river and A and B are two islands in the river

This is the only impossible task that was given to GPS. Although GPS's behavior is not aimless (it crosses six bridges in two different ways), GPS cannot see the impossibility because it lies in the topological properties of the bridges. GPS only attempts to cross bridges and has no way of viewing the problem as a whole.

Water jug

Given a five gallon jug and an eight gallon jug, how can precisely two gallons be put into the five gallon jug? Since there is a sink nearby, a jug can be filled from the tap and can be emptied by pouring its contents down the drain. Water can be poured from one jug into another, but no measuring devices are available other than the jugs themselves.

For the water jug task, means-ends analysis seems to be a rather ineffective heuristic as demonstrated by the fact that GPS stumbled onto the solution. Better types of differences would improve GPS's performance on this task. Better differences do exist although they are quite complex (involving modular arithmetic, naturally).

Letter series completion

This task, which is found in aptitude tests, is to add the next few letters to a series of letters. GPS correctly completed the series,

B C B D B E - - .

The letter series completion task is the only task whose solution requires inductive reasoning. The formulation is quite clumsy, but this example demonstrates how the problem can be approached by searching for a suitable description in a space of descriptions. The binary choice task, another task requiring inductive reasoning, was formulated in a similar way by Feldman, *et al.*[25] (Simon and Kotovsky[26] use a different formulation of letter series tasks.)

Generality of GPS's basic processes

The appeal of the heuristic search paradigm lies in its generality. Hence, it is important that GPS's problem solving techniques have general applicability to heuristic search problems.

GPS uses all of its basic processes (described above) except for *operator-difference*, on each of the eleven tasks. In solving the predicate calculus, three coins and letter series tasks, GPS never attempts to apply an infeasible operator, and thus the *operator-difference* process is never evoked. If each of the basic processes were specialized to a particular aspect of one task, it would be necessary to add new basic processes in order to give GPS a new task. However, the applicability of the basic process to all eleven tasks implies that they would also be applicable to other tasks.

REFERENCES

1 G W ERNST A NEWELL
Generality and GPS
Carnegie Institute of Technology 1966

2 A NEWELL
Self-Organizing Systems
Some problems of basic organization in problem-solving programs
Yovits M C Jacobi G T and Goldstein G D eds
Spartan Books Washington D C 1962 p 393-425

3 A NEWELL
A guide to the general problem-solver program GPS-2-2
RAND Corporation Santa Monica Calif RM-3337-PR 1963

4 A NEWELL
Proceedings of IFIP congress 62
Learning, generality and problem-solving
North Holland Publishing Amsterdam Holland 1962

5 A NEWELL J C SHAW H SIMON
Preliminary description of general problem-solving program –I GPS-1
CIP Working Paper No 7 Graduate School of Industrial Administration Carnegie Institute of Technology Pittsburgh Pa Dec 1957

6 A NEWELL H C SHAW H A SIMON
Information processing: proceedings of the international conference on information processing
Report on a general problem-solving program for a computer
UNESCO Paris 1960 p. 256-264

7 A NEWELL H A SIMON J C SHAW
Self-organizing systems
A variety of intelligent learning in a general problem-solver
Yovits M C and Cameron S eds
Pergamon Press New York 1960 p 153-189

8 A NEWELL H A SIMON
Science
Computer simulation of human thought 134
P 2011-2017 December 1961

9 A NEWELL H SIMON
Lernende automaten
GPS a program that simulates human thought
Munich Germany 1961

10 H A SIMON A NEWELL
Current trends in psychological theory
The simulation of human thought
University of Pittsburgh Press Pittsburgh Pa p 152-179 1961

11 H A SIMON A NEWELL
Management and the computer of the future
Computer simulation of human thinking and problem solving
Greenberger M ed
John Wiley and Sons New York 1962

12 H A SIMON A NEWELL J SHAW
Contemporary approaches to creative thinking
The processes of creative thinking
Gruber H A Terell G and Wertheimer M eds
Atherton Press New York p 63-119 1962

13 D G BOBROW
Natural language input for a computer problem solving system
Doctoral Dissertation Mathematics Dept Massachusetts Institute of Technology Cambridge Mass 1964

14 B RALPHAEL
SIR: A computer program for semantic information retrieval
Doctoral Dissertation Mathematics Dept Massachusetts Institute of Technology Cambridge Mass 1964

15 A NEWELL G ERNST
Proc of IFIP congress 65
The search for generality
Kalenich W A ed
Spartan Books Washington D C p 17-24 1965

16 A NEWELL ed
Information processing language-v manual
Prentice Hall Englewood Cliffs N J 2nd ed 1961

17 J R SLAGLE
J ACM 10
A heuristic program that solves symbolic integration problems in freshman calculus
p 335-337 Oct 1963

18 HORMANN AIKO
Behav sci 10
Gaku: an artificial student
p 88-107 Jan 1965

19 J A ROBINSON
J ACM 12
A machine-oriented logic based on the resolution principle
p 23-41 Jan 1965

20 A NEWELL J C SHAW H SIMON
Empirical explorations with the logic theory machine
Proceedings of the Western Joint Computer Conference
p 218-239 1957

21 P C GILMORE
IBM j res develop
A proof method for quantification theory
p 28-35 Jan 1960

22 J McCARTHY
Situations actions and causal laws
Stanford Artificial Intelligence Project Memo No 2
July 1963

23 J McCARTHY
Proc symposium on mech of thought processes
Programs of common sense
Her Majesty's Stationery Office London p 75-84 1959

24 FISHER BLACK
A deductive question answering system
Doctoral Dissertation Division of Engineering and Applied Physics
Harvard University Cambridge Mass 1964

25 J FELDMAN F TONGE H KANTER
Symposium on simulation models
Empirical explorations of a hypothesis-testing model of binary choice behavior
Hogott A C and Balderston F E eds
South-Western Publishing Co Cincinnati Ohio p 55-100 1963

26 H A SIMON K KOTOVSKY
Psychological review 70
Human acquisition of concepts for sequential patterns
p 534-546 June 1963

A compact data structure for storing, retrieving and manipulating line drawings

by ANDRIES Van DAM
Brown University
Providence, Rhode Island

and

DAVID EVANS
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

The field of graphical man/machine interaction is customarily split into hardware and software areas. The former can be considered to have come of age: there are over twenty-five brands of off-the-shelf consoles with all the requisite input devices, and new techniques and improvements are constantly being developed. Many consoles are also provided with primitive supporting software which allow one to draw points, lines, arcs, etc., in a symbolic language of some sort. Less well understood and developed, however, is that aspect of display software concerned with representing and manipulating the problem model from which these primitive point/line/arc pictures are derived. The "data structure" is the machine representation of the often complex and hierarchical problem model. It must be judiciously derived from the model on the one hand and, on the other, lead readily to the reduced console display file of points, lines and arcs which cause the actual visual display. Furthermore, the data structure must be efficiently stored and processed (usually contradictory requirements).

Historically, pictorial (line drawing) data structures have been at one of two extremes: the obvious, first-approximation, single-level structure already on the point/line/arc-level of the console display file, or a highly complex, hierarchical and interconnected list structure (Ross's "plex,"[1] Sutherland's "ring,"[2] Roberts' CORAL lists,[3] or even Feldman's[4] and Rovner's[5] hashing schemes for simulating an associative memory). The second method, in our opinion, is more elegant and powerful in terms of storage and processing economy for hierarchical and repetitive pictures. However, the degree of interconnectedness of the list structures (forward pointers, backward pointers, head-of-list and tail-of-list pointers, etc.) may considerably inflate the size of the block of storage (the "item") which represents a picture. In addition, most of the above schemes rely on fixed position of the various pointers within blocks, so that the growth of an item is relatively constrained, and updates are intricate. Special list processing languages (such as CORAL) in which one can write picture processors have been written to manipulate these data structures and pointers.

An alternate approach is described below. The data structure is purposely kept as free of pointers as possible (without losing any topological or geometrical information) to reduce the size of a given item to the absolute minimum, and picture processing "primitives" are written as "worker programs" in assembly language or are bootstrapped from simpler ones. The subpicture hierarchy is recursively built into every "master" item by describing it as being composed of points, lines, and "instances" of lower-level subpicture masters, to which affine transformations have been applied. The MULTILANG system* provides automatic linking and retrieval of those items using instances of the same subpicture master, and of the primitives' worker programs. The IBM 7040 implementation described below accepts card input, and uses high-speed printer output since a vector scope is not available. Input of all commands is therefore in a digital rather than a light pen/function key language. Pictorial ENCodIng Language (PENCIL) primi-

*"THE MULTILANG on-line programming system", R.L. Wexelblat and H.A. Freedman. Proceedings of the 1967 Spring Joint Computer Conference.

Table I

PENCIL Primitives*

A. DEFINITION

* POINT/NAME/X,Y/OP — where OP = N, for NAME, and T, for TEXT

* LINE/NAME/PT1/PT2

ARC/NAME/PT1/PT2/PT3

PARC/NAME/F/V/PT1/PT2

EARC/NAME/F1/F2/PT1/PT2

B. MANIPULATION (level 1)

* COIN/PT1/PT2

* MERGE/PT1/PT2

* COPY/NAME/L/PT

* INTPT/NAME/L/X,Y

SPIN/L1/L2/PT/DEGREE

C. TRANSFORMATION

* MOVE/NAME/OP/PT/QUAL — where OP = P, for point; N, for node; S, for subpicture; D, for display, i.e., the screen level picture

* ROTATE/NAME/OP/PT/DEGREE/QUAL

* SCALE/NAME/OP/PT/DEGREE,XX,YY/QUAL

D. CONTROL

* CLEAR

* DELETE/NAME/OP/QUAL — where OP = P, for point; N, for node; L, for line; S, for subpicture; U, for uniformly

* ASSIGN/NAME/NODE1/.../NODEn

* USE/NAME/QUAL

* SHOW/NAME

LOCATE/X,Y; ENDLOC

TEXT/PT/OP1/H/W/OP2/-------$-------$...$$

where OP1 = TL, for top left; TR, for top right; BL, for bottom left; BR, for bottom right

OP2 = L, for left; R, for right

*Those primitives included in the present version of PENCIL are identified by *. This set provides all basic capabilities, including hierarchical assembly of pictures.

```
XTRANS PROC    BUFIN
       CLEAR
       USE/DIODE/1
       USE/DIODE/2                  LINE/L4/P4/P6
       USE/DIODE/3                  LINE/L5/P7/P8
       USE/RESIS/4                  POINT/P10/=15,=0
       USE/RESIS/5                  LINE/L6/P6/P10
       USE/RESIS/6                  LINE/L7/P10/P9
       USE/RESIS/7                  MOVE/DG1/N/P1/1
       USE/TRANS/8                  MOVE/DG2/N/P2/2
       USE/CAPAC/9                  MOVE/DG3/N/P3/3
       USE/GROUND/11                MOVE/A4/N/P3/4
       POINT/P1/=-21,=24            MOVE/A5/N/P9/5
       POINT/P2/=-21,=12            MOVE/D28/N/P9/6
       POINT/P3/=-21,=0             MOVE/B6/N/C8/6
       POINT/P4/=-15,=12            ROTATE/RESIS/S/A7/=90./7
       POINT/P5/=-15,=0             MOVE/A7/N/P5/7
       POINT/P6/=-12,=12            MOVE/D29/N/P6/9
       POINT/P7/=12,=12             MOVE/A11/N/C8/11
       POINT/P8/=15,=12             ASSIGN/BUFIN/IN1/IN2/IN3/OUT
       POINT/P9/=21,=0       END
       POINT/OUT/=57,=12     XBEGIN
       POINT/IN1/=-48,=24
       POINT/IN2/=-48,=12
       POINT/IN3/=-48,=0
       LINE/L1/P1/P3
       LINE/L2/P3/P5
       LINE/L3/P5/P4
```

Figure 1 – Procedure BUFIN

tives (see Table I) are specified as MULTILANG statements, and picture processing "procedures" composed of these statements are processed by MULTILANG. (Figure 1, procedure BUFIN, is a sample procedure† for the highest-level drawing indicated schematically in the exercise of Figure 2; Figure 3 is its result on the printer. Figure 4 shows the same circuit drawn from a points-and-lines model on the IBM 2250). The PENCIL picture processor itself is not dependent on these digital modes of I/O: only small pre-and post-processing I/O routines need be altered to accommodate a vector scope, while the digital inputs were chosen to be as similar as possible to scope-mode inputs.

A complete specification of the system described below may be found in Reference 7.

PENCIL Items

Assumptions

In the PENCIL approach several fundamental assumptions were made. One was that the hierarchical subpicture structure is natural for the class of two- or three-dimensional line drawings under consideration (flowcharts, block diagrams, electrical circuit schematics, trusses, pipe diagrams, etc.). This led to the SKETCHPAD[2] master/instance technique. It was also assumed that picture items should be stored and retrieved in the same manner as are any other types or data or program in the MULTILANG

†Two small discrepancies between the format of POINT here and POINT in Table 1 exist: x,y values are passed with a Hollerith =sign, and OP = N for NAME is assumed when left blank.

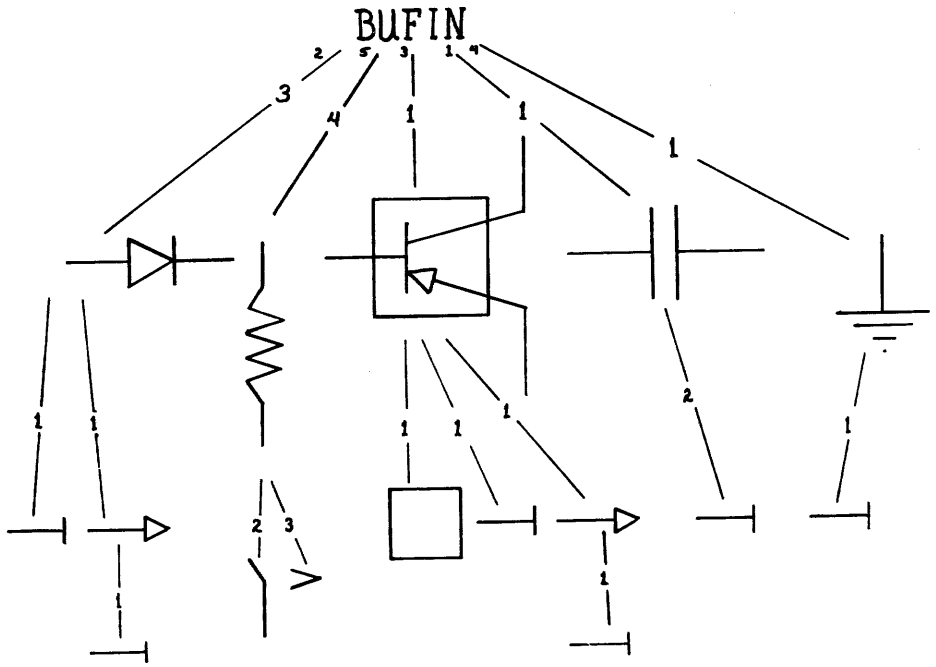


Figure 2 – Assembly tree of BUFIN

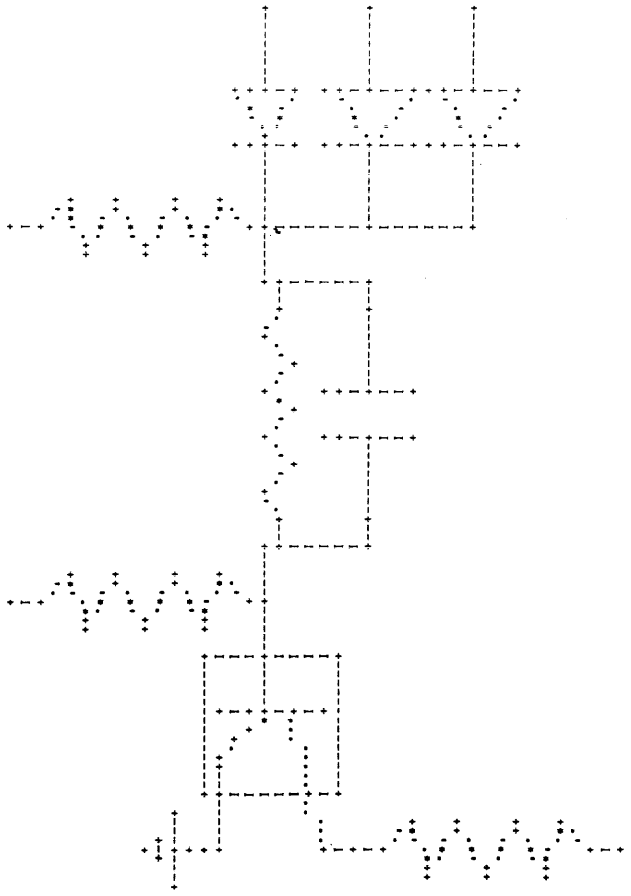


Figure 3 – Buffer inverter drawn on 1403

system: by "description," i.e., by an essentially Boolean function of key words (keys) or ranges of keys. Thus browsing is possible, since a picture or pictures can be identified by a variable configuration of keys pertaining to topological, geometric or merely descriptive information (e.g., number or type of components used, date on which it was drawn, manufacturer(s), power rating of a specific component, etc.). The third (and easily accepted) assumption was that core and disk space were at a premium, especially if realistically large files of items containing much "descriptive" information were to be handled.

These assumptions led to the use of MULTILANG threading and retrieving as a basis for PENCIL, since it is feasible to avoid redundancy and still get all the necessary information from the data structure without undue sequential searching.

Control item and line item

The basic unit of data is a picture, composed of points, lines and/or other pictures. The PENCIL primitives operate on these pictures to DEFINE, MANIPULATE, TRANSFORM or CONTROL them (see Table 1). Any picture may be used recursively as a component in a "higher-level" picture, and at such time is designated a "subpicture."

A picture (subpicture) is represented in the file by a "Control Item" and associated "Line Items," "Text Item," and "Information Item." Each item is a MULTILANG data item[6] and as such has the control information which MULTILANG requires. Figure 5 shows schematically a completed Control Item, the principal picture item containing point and subpicture information. The MULTILANG interpretation of the blocks is listed on the left and the PENCIL interpretation on the right (see below).

The Control Item contains a first, "primary" key (and its link) which is initially set to a non-printable internal code and is replaced with the assigned name of the picture when the user completes it. Following four special-purpose keys and their links (see Appendix A, section 1), keys occur which are the names of the subpictures composing the picture. The geometrical disposition of each subpicture within the picture is represented in the Control Item by a data element which contains a cumulative record of all transformations which were applied to the subpicture. This "Transformation Matrix" data element (TM) contains a rotation/scale matrix and a translation vector for each occurrence (instance) of the given subpicture. There is also single "Proper Transformation Matrix" data element (PTM) which accumulates the transformations applied to the screen (current display) as a whole (including all subpictures, points and lines of which the screen is composed).

Points explicit on the screen are represented in the Control Item in a "Points Table" data element (PT) in which the name of a point and its x,y,(z) values on the screen are listed consecutively. Lines defined on the level of the screen are not explicitly contained in the Control Item: they are encoded in separate MULTILANG Data Items called "PENCIL Line

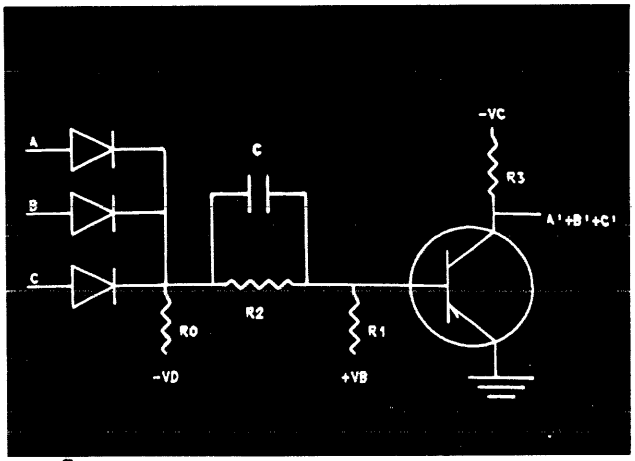


Figure 4 – Buffer inverter drawn on 2250

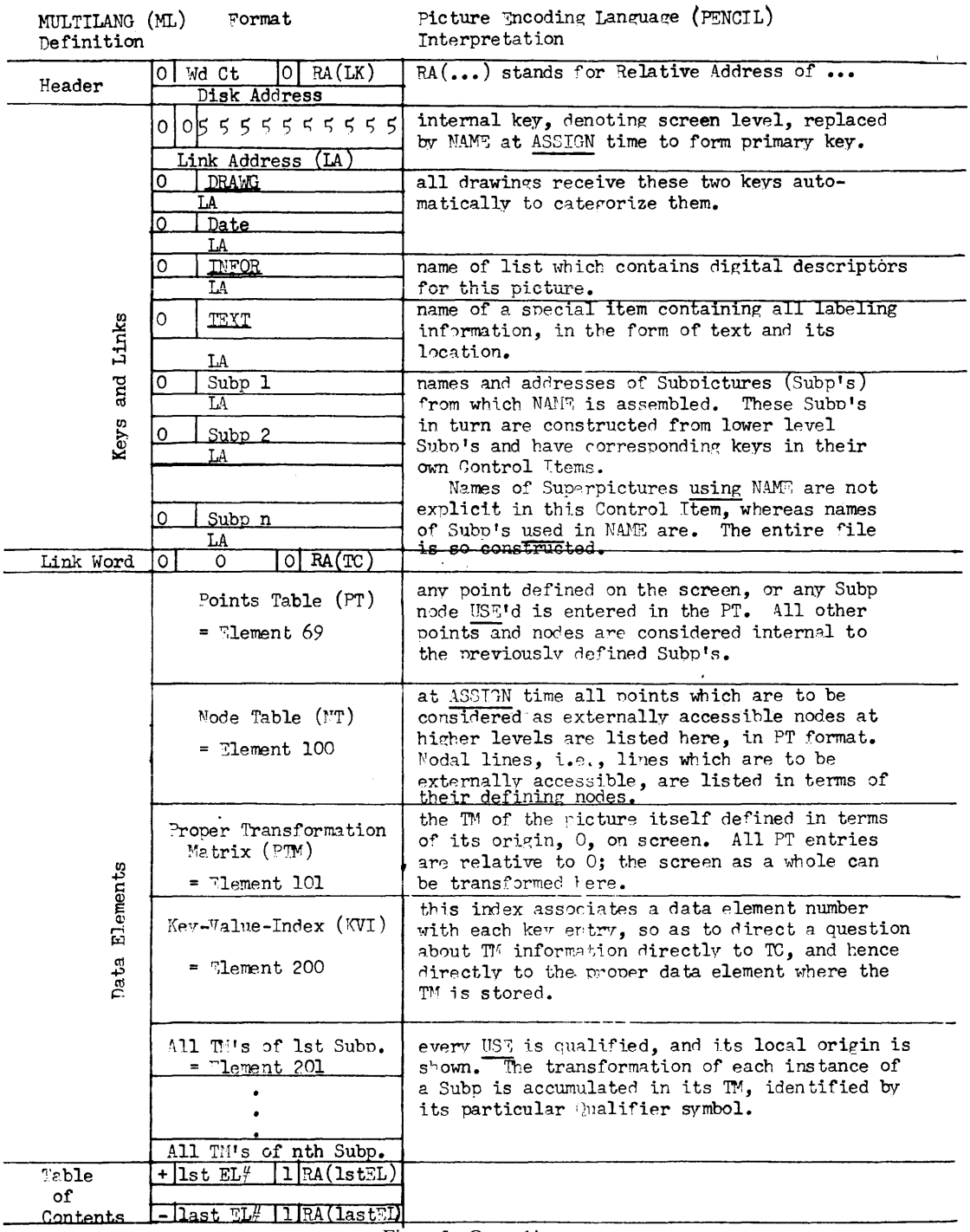


Figure 5 – Control item

Items"* (Figure 6) in which they are defined in terms of their endpoints. These endpoints are keys in the Line Item and also entries in the Points Table of the super Control Item. The Control Item is linked to all its Line Items through the appearance of its picture name as a key in all of them. The primary key of a Line Item is the key *LINE*†† and its name is a second key. Thus, data items can be uniquely categorized: if a picture name appears as primary key in the item, it is the Control Item for that picture. If the picture name is not the primary key and if the key *LINE* is present, it is a Line Item belonging to the

*In what follows "line" or "Line Item" denotes both a conventional straight line (*LINE*) and a conic section: circular arc (*ARC*) parabolic arc (*PARC*) or elliptical arc (*EARC*). The corresponding Items are similar to the Line Item.

††Italized words of capitals such as *LINE* or *CLEAR* denote the identical alphanumeric characters strings in PENCIL Items and primitives. Words in capitals but not italized denote variable names.

ML	Format				Picture Encoding Language (PENCIL) Interpretation
Header	0	17	0	12	
	Disk Address				
Keys & Links	0	LINE			LINE as primary key identifies this as a Line item, as opposed to a picture control item.
	Link Address (LA)				
	0	5 5 5 5 5 5 5 5 5 5			replaces at ASSIGN time by name of picture in which this line is used.
	LA				
	0	NAME			the line's own NAME
	LA				
	0	Endp 1			endpoints in terms of which the line is defined
	LA				
	0	Endp 2			
	LA				
Link Word	0	0	0	15	
Data Elements	RA(endp 1) in Control Item's PT				
	RA(endp 2) in Control Item's PT				
Table of Contents	+	1	1	13	
	-	2	1	14	

Figure 6 – Line item

picture; if the key *LINE* is absent, the item is the Control Item of a superpicture using the subpicture with the specified name.

Keys and subpicture structure

The choice of the function or interpretation of keys is determined by the method of storing the hierarchical structure of pictures. First of all, it is usually advantageous to assemble as much as possible of the drawing on a subpicture level. Hence subpictures should be included in Control Items very efficiently and compactly. A threaded list structure using the names of subpictures of a given picture as keys would seem the obvious answer, but the utility of this type of a file should be compared with the advantages of an "inverted" file in which each subpicture would contain, as keys, the names of superpictures which use it. In terms of space required the two kinds of files are identical. In terms of processing time, they are not: the question "where is this picture used, and how many times?" (a typical parts inventory question) is asked much less frequently than "of what is this picture composed?" For instance every time a change is made on the screen, the entire component tree, or at least a branch of it, must be traversed to yield the subpicture structure explicitly; this disassembly process would be much less efficient for the inverted filestructure.

To produce the console display file with the present scheme, a canonical scan with pushdown is performed of the tree of subpictures of the current Control Item. Each node of the tree corresponds to a picture instance: the terminal nodes represent pictures of points and lines only ("level 1") and the root node represents the current screen. Each picture is printed recursively, by printing its subpictures and then its screen level points and lines. Therefore, the first item to be completely printed is the left-most and down-most "level 1" subpicture of the tree.

Matrix pre- and post-multiplication takes place at every node in order to adjust for the current geometrical disposition of the corresponding subpicture within the higher level. Subpicture Control Items are retrieved by MULTILANG from disk or core, as is appropriate. The item found is then placed in a pushdown stack for further reduction, while a copy is maintained in core (if space is available) to avoid another

disk retrieval in case another instance of this subpicture master item is required elsewhere in the tree.

With this choice of key structure the problem of locating a specific line or lines for output or for manipulation is handled by initiating a MULTILANG retrieval on the parent picture name, *LINE*, and the name of the point(s), locating all appropriate lines in sequence; given a line, finding all connecting lines involves initiating a retrieval on the parent picture name, *LINE*, and the names of the endpoints in the first line, etc. Such properties as topological intersection and common use of elements are easily handled in a similar manner.

In addition to the output sequence advantage described above, another good reason for not choosing the "inverted" file is that updating (and hence retrieving and manipulating) of the used subpicture would be required every time the using subpicture added or deleted it. With the built-in component tree, on the other hand, changes are made only to the Control Item and neither to its subpictures nor to its superpictures. Duplication is wasteful of storage, and so it was decided neither to incorporate both files simultaneously nor to store one file structure in a special data element of a picture item. The "inverted" file can be quickly obtained from the component file by retrieving all items with the subpicture's name as key: a single retrieval request in MULTILANG produces sequentially all such items, since they are linked. For each of these items all their super items can be retrieved, etc.

Appendix A contains additional information on the structure of the Control Item.

PENCIL primitives and the assembly process

PENCIL processes take the names of subpictures and lines as data and manipulate the corresponding Control Items and Line Items. The processes can be considered as primitives in an open-ended, growing language, in much the same way that IPL-V's[8] primitives, the "Js," are considered. The intent is to provide a handful of efficient and basic primitives which constitutes a sufficiently complete set to allow bootstrapping, again in the same manner as that in which the IPL-V "J"s are bootstrapped. Calling PENCIL primitives by MULTILANG statements makes it possible to form MULTILANG "procedures" to assemble pictures, or in fact to build higher-level primitives, just as higher-level pictures are built. The eventual capability of MULTILANG to make hierarchical procedure calls and to perform subroutine calls with substitutable parameters will allow the indefinite nesting of PENCIL primitives for efficient bootstrapping.

The set of primitives listed in Table 1 is divided into four subsets, roughly according to similarity of function. The DEFINITION verbs serve to establish new data in the form of points, straight lines or conic sections. The MANIPULATION verbs move lines and points on the screen by making points coincident or identical through merging, and by rotating lines to make specified angles with other lines. TRANSFORMATION verbs are used to apply affine transformations to subpictures used on the screen. The CONTROL verb *CLEAR* erases the PENCIL working area, initializes the screen, and assigns blocks of unused storage to form a skeletal Control Item (and a Text Item which stores digital annotation/legend information for the drawing). *ASSIGN* names the subpicture which has been assembled on the screen and sets up MULTILANG and PENCIL control words in the Control Item prior to calling on MULTILANG to store the item on the disk. *USE* retrieves a picture stored on the disk and displays it on the screen as a subpicture by putting the relevant control information in the Control Item of the screen picture. *SHOW* simply displays the specified picture. *DELETE* erases points, lines and specific subpictures from a given picture, and also can be used to delete a picture uniformly in the file. LOCATE models the pointing feature of light pens.

Appendix B contains additional information on some of these primitives.

CONCLUSIONS

The 7040 PENCIL system has run satisfactorily with only 7000 36-bit words (42K characters) available for both programs and pictorial data, exclusive of MULTILANG. Real time response with a light pen and a buffered scope would require more core for reasonably complex pictures. Pictures frequently used should probably be reduced to points and lines to eliminate tree scanning.

Some other properties of the system are listed below:

(1) Subpictures may be added and deleted easily, in any order; items are of variable size, and there is no order among keys or data elements.

(2) Pictures are intermixed in memory with other types of data and programs. A picture can be treated exactly as is any other block of information, or can be singled out and subjected to special picture processing. In particular, browsing through the pictures with even rather vague criteria is very easily (and economically) done, whereas in most other systems only retrieval by identification number only is possible.

(3) Since pictures are manipulated and drawn by executing statements and procedures in an interpretive mode, picture-processing statements may be mixed

with retrieval or computation statements in the same procedure.

(4) The overhead per subpicture is as low as possible, no more than twelve words for the two-dimensional case: key and link, identifier plus six-word transformation matrix, and at most three additional control words. [Nodes (see Appendix A.2), of course, are optional.] With a reasonable amount of core, most of PENCIL and the entire current picture could be kept in core, especially if it were kept in its reduced points-and-lines form. Only light pen pointing would make this reduction impractical. Some schemes for keeping part of the picture in tree form and part of the picture in reduced form are being considered for the System/360 implementation.

(5) Pictures might be stored more compactly as the PENCIL procedures which drew them than as the equivalent Items (providing, of course, that they were drawn efficiently and without too much trial and error).

ACKNOWLEDGMENTS

The work described in this paper was carried out at the University of Pennsylvania, primarily through support from the Research Division of the Naval Bureau of Supplies and Accounts, and the Information Systems Branch of the Office of Naval Research, under Contract NOnr 551(40).

REFERENCES

1 D T ROSS and J E RODRIGUEZ
Theoretical foundations for the computer-aided design system
Proceedings of the 1963 Spring Joint Computer Conference Vol 23 p 305 1963

2 I E SUTHERLAND
Sketchpad A man-machine graphical communication system
Lincoln Laboratory Technical Report No 296 1963

3 W R SUTHERLAND
The on-line graphical specification of computer procedures PHD dissertation submitted to MIT January 1966. See also
W. KANTROWITZ
CORAL macros-reference guide
Lincoln Laboratory Technical Memorandum No 23L-0003 1965

4 J A FELDMAN
Aspects of associative processing
Lincoln Laboratory Technical Note 1965-13 1965

5 P D ROVNER
An investigation into paging a softwares–simulated associative memory system
University of California (Berkeley) Document No 40.10.90 1966

6 R L WEXELBLAT and H A FREEDMAN
The MULTILANG on-line programming system
Proceedings of the 1967 Spring Joint Computer Conference

7 A van DAM
A study of digital processing of pictorial data
Ph D dissertation Department of Electrical Engineering University of Pennsylvania 1966

8 *Information processing language–V*
Prentice-Hall 1964

APPENDIX A – Additional Information on the Control Item

A.1 Keys and auxiliary information storage

In addition to the primary name key and the subpicture keys, there are four standard keys entered initially in each Control Item (see Figure 5). *DRAWG* is a key which all Control Items share to distinguish them from other MULTILANG items in the MULTILANG file. *DATE* is obtained from the IBSYS 7040 Operating System and allows retrieval of all drawings made within a certain time span. *INFOR* is a key of a list of standard MULTILANG data items, exactly one such list being keyed to any one picture by that picture's name. An *INFOR* item contains all manner of digital descriptive information which one may wish to store in order to be able to retrieve subpictures not merely by their "pictorial" (i.e., component) structure. It also contains physical component characteristics, test data, etc. Applications programs would use MULTILANG item definition programs to establish these items. Thus the Control Item contains only structural (pictorial) information.

The fourth key is *TEXT* and links a Text Item to each picture. This item contains all annotation information in the form of text "boxes": all text is considered as written on successive lines in a rectangular enclosure (never externally apparent) which is "tacked" to the picture at one of its corner points. If the corner point is defined within a subpicture, the text box will be moved with its corner point as the subpicture is moved. The internal format of a Text Item is a set of data elements, one per text box, containing (1) the segmented text and (2) a control word giving the dimensions of the text box, the corner point which serves as the anchor, and whether the text is to be left- or right-justified within the box. Individual points on screen level may also have associated with them a single label, their name, as specified in the option of *POINT*.

A.2 Points and node table (PT and NT)

On the current display level there exist two types of points: those defined at that level and those at subpicture level. Those at subpicture level are in the current display because they were declared "nodes" at the time the subpicture was *ASSIGN*ed. Only such points are considered accessible (in the sense of connections) on the current level; all others are considered internal to the given subpicture. The same point may be declared a node in as many as five different levels. In both tables, point names alternate

with their x,y (z) values. Flags are used to indicate whether the point is to have its name displayed or whether it is the anchor point of a text box.

A.3 Proper transformation matrix (PTM) and sub-picture transformation matrices (TM's)

After a display has been partially filled it may be desirable to move, rotate or scale it in its entirety. The screen has a Transformation Matrix of its own, a "Proper" Transformation Matrix (PTM), which accumulates affine transformations of the entire screen. Furthermore, to each subpicture in a Control Item there corresponds one Transformation Matrix data element in which are stored the Transformation Matrices (TMs), one per instance, of the given subpicture. The entire display or individual subpictures may be transformed in an arbitrary sequence. The results are accumulated in the appropriate elements.

A.4 Key value index (KVI) and table of contents (TC)

It is often necessary that primitives be able to manipulate Transformation Matrices and subpicture-name keys. There must therefore be a quick way to map from a key to the data element containing the corresponding Transformation Matrices. The Key Value Index (KVI) associates an element number from a strictly increasing integer sequence with each new subpicture name key.

The Table of Contents is a list of pairs; the first part of each pair is a data element number, and the second part is the Relative Address (RA) of the first word of that data element. The Table of Contents of an item containing subpictures always contains the four standard entries 69 (PT), 100 (NT), 101 (PTM), 200 (KVI), and as many other entries as there are subpicture name keys in the Control Item. The Table of Contents and the KVI are used to find the Transformation Matrix data element for a given instance of a given subpicture-name key.

APPENDIX B—Control Primitives

The pictorial assembly process using PENCIL primitives is initialized with a *CLEAR*. The drawing of points, lines or subpictures on the blank screen may then proceed.

As soon as the picture is *ASSIGN*ed, it is ready to be used as a subpicture on a higher-level screen. In assembling a complicated drawing, the user must compromise between the tediousness of constructing the picture entirely on the points-and-lines level, and the comparative ease of constructing it on the subpicture level (which makes the drawing more compact but slower to retrieve and process). For example, standard electrical symbol "macros" such as resistors and capacitors are best drawn on the points-and-lines level to avoid retrieval of one or two additional levels of subpictures. An entire buffer inverter, however, is more easily and economically drawn using at least one level of macros.

Since a given subpicture may be *USE*d a number of times on the screen, particular instances must be uniquely identified. This is accomplished by "qualifying" each separate instance of a subpicture. Each *USE* (one per instance) must specify a unique one-or two-character qualifier to be suffixed to all labeled points of the given subpicture. Each picture is assembled on screen level with respect to the screen orgin (0,0) which becomes its "local origin" if the picture is used as a subpicture. Since this local origin of the subpicture instance, properly qualified, always appears with it on the screen and in the Points Table of the screen level Control Item, there is a unique differentiation between instances. Each instance, furthermore, has a separate entry in the Transformation Matrix data element of the subpicture in the screen level Control Item.

During the assembly of a picture, manipulations defined on lines and points are intermixed with those on retrieved subpictures, and there is often occasion to connect subpictures by screen level lines. When the picture is finished and is to be stored (*ASSIGN*ed), the user may specify a list of those screen level points (level 1 and/or subpicture level) which are to be considered as nodes, to allow attachment in higher level usage. He may then "attach" to lines by attaching to the nodes which define them. When the subpicture is *USE*d, only points designed as nodes and the local origin will appear explicitly and in qualified format on the screen.

CLEAR initializes the screen level Control Item with an internal primary key of $(5555555555)_8$. The user names a finished picture with *ASSIGN*, which replaces this "screen" key by the NAME operand in all screen level items. There is never more than one screen level picture, and hence never more than one set of related items with the screen key as a key. As long as the screen key remains, components may be freely manipulated; once the screen is *ASSIGN*ed (and therefore stored), it becomes an immutable entity with a fixed topology, and a geometry which may be changed only by affine transformations on the higher-level screens utilizing this picture. If a user wants to change the structure of a stored picture, he must retrieve it by *SHOW*, which is the inverse of *ASSIGN* in that it replaces the primary key of the Control Item and the picture-name key in all corresponding Line Items with the screen

key, and in effect reinitializes the assembly process on that picture.

An important feature of the assembly process is that it may be mixed with non-pictorial manipulations and computations—the primitives are completelv context-free due to the nature of MULTILANG procedures and their execution.

Experience using a time-shared multi-programming system with dynamic address relocation hardware

by R. W. O'NEILL
IBM Watson Research Center
Yorktown Heights, New York

INTRODUCTION

The IBM Research M44 computer is an experimental machine which was installed in the Thomas J. Watson, Sr. Research Center, in Yorktown Heights, New York, in November of 1964. The machine is an extensively modified IBM 7044, a binary single-address fixed-word-length computer with a CPU cycle time of 2 micro-seconds.

The machine was designed by Research personnel with two goals in mind. First, there was a need for a large-scale computer which could be approached at the hardware and software system levels for various kinds of experimental activities. Second, it became important to implement a paging machine and measure its performance under a variety of workloads.

The M44 has become a full computer system which is time-shared by users in the building and around the country. It is a multi-programmed system which is also used to do batch processing. There are a number of experimental software systems which are implemented and play an important role in the work of various departments of the building. And there are several special purpose hardware devices attached through the direct data channels of the machine to facilitate certain other projects.

Among the more interesting applications are:

- a project in the automated design of computers.
- a computer operating system study.
- a project relating to the design and use of terminals in an interacting environment.
- the utilization of special hardware to test monolithic circuits.
- a hardware/software complex which controls traffic in one of the tubes of the Lincoln Tunnel.
- a collection of activities, mainly under an ARPA contract, in the use of such software systems as IPL5, LISP, and SNOBOL.

There is in addition much day-to-day work of a conventional kind, and, what is perhaps the most interesting aspect of the project, the measuring and evaluation of the entire complex of hardware and software.

The intent of this paper is to present the results, to date, of the measuring and evaluating activity. A necessary part of such a presentation is a description of what is being measured. We believe that the characteristics of this paging machine, are translatable, with scaling, to other paging machines and should therefore be of some interest to persons working in this area.

M44 hardware

The M44 computer is a highly modified version of the IBM 7044. It is a single address computer operating with 21 bit addresses, with the instruction format, index registers, and the instruction counter modified to handle 21 bit addresses. It has 32,768 words of 2μsec computational store, plus an additional 196,588 words of 8μsec computational store. The M44 has 3 modes of operation which are not found on the standard 7044: Problem/Supervisor mode, Location Test/no Location Test mode, and Mapping/no Mapping mode. The Problem/Supervisor mode capability is the rather conventional facility of reserving certain instructions, notably I/O and mode changing instructions, for execution only in the supervisor mode. The Location Test Mode in conjunction with the Location Test Register provide a convenient debugging tool which can be used to selectively halt the execution of a program when reference is made to any specified storage cell. The Mapping mode provides a dynamic address relocation mechanism which may be used to solve the problems of storage allocation in a multi-programmed environment.

The mapping device of the M44 computer consists of a 16,384 word 2μsec memory and its associated logic. It operates in the following way—the 21 bit addresses generated in the M44 CPU are broken into 2 parts, say the leading 14 bits and the least significant 7 bits. (See Figure 1). The leading 14 bits are then used

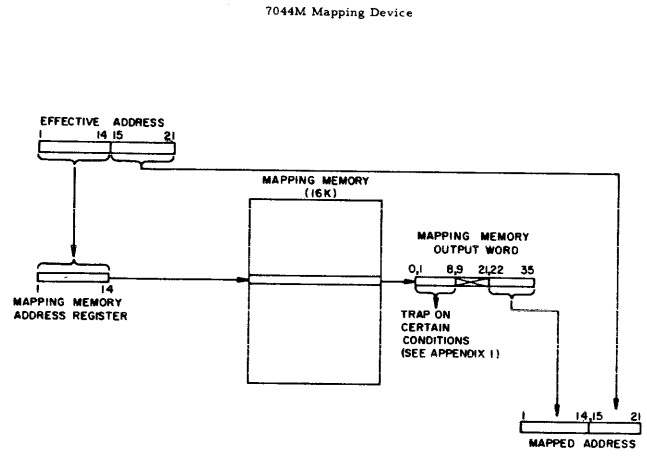


Figure 1

to address the 16K mapping memory directly. A 14 bit field of the mapping memory word addressed is then concatenated with the 7 bit field of the logical or unmapped address to create a physical address which is now used to access the core memory. This is the basic mapping procedure. However, there are some refinements to the procedure implemented on the M44. First by concatenating the leading bits of the unmapped address with the contents of a fixed register, called the ID register, it is possible to have the same logical address map into distinct physical addresses without changing the contents of the mapping memory. (See Figure 2). Only the ID register

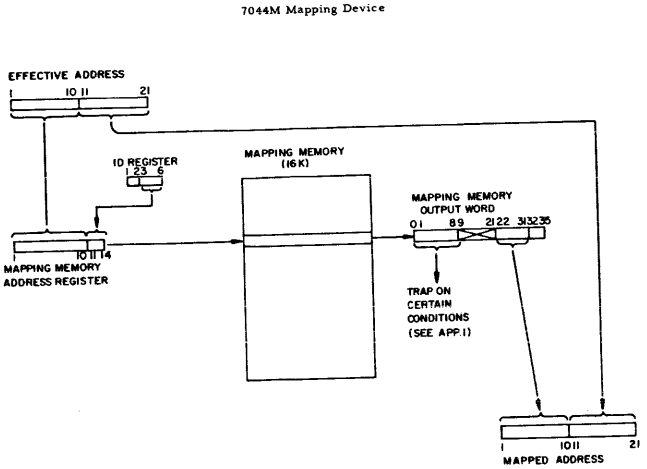


Figure 2

must be changed. Of course since the 16K mapping memory requires a 14 bit field to address it, the sum of the leading bit field and the ID field lengths must always be 14. Therefore one might choose 10 and 4 bit field lengths respectively. The reason for calling this additional register the ID register is probably already apparent. By changing it appropriately when switching the CPU between virtual machines it can be used by MOS to distinguish between the pages of core assigned to distinct 44X's. The second refinement is required due to the fact that although 2 million distinct logical addresses can be generated (2^{21}) only 229,376 words of physical core are available on the M44. Therefore one bit of a separate field (called the status bit field) of the addressed mapping memory word is interrogated by the hardware to verify that this is an address which is currently mapped into a physical address of the M44. If it is not currently mapped, a trap to the supervisory program is generated. The supervisory program can then take appropriate action, for example it might initiate a transfer of the logical page from the back-up store (the 1301 II disk) to a physical core location. Upon completion of this transfer it could modify the mapping word accordingly. Because of the ready availability of the status bit field for interrogation at every memory reference, these bits are used to implement a hierarchy of storage protection spheres in the M44 and are also used for data gathering to assist the experimental use of the M44. A complete listing of the status bits is found in Figure 3. This particular hardware implementation of the address translation function contains two drawbacks which would not be acceptable on a production system. The number of virtual machines (44X's) which can be identified is a function of the page size, and the 2μ sec mapping time is added to every memory cycle. Since the fundamental translation function is unchanged, for the purpose of experimentation these drawbacks are more than compensated for by the ease of construction of the experimental system.

The I/O and storage configuration of the M44 is shown in Figure 4.

44X (virtual machine) description

One can imagine a more or less ideal computer, or *virtual* machine closely related to the M44. It has two million words of core. It has in effect a separate channel for every I/O unit. It has an instruction set which differs from that of the M44 because it has no mapping device and no problem mode. It does take traps and interrupt for a variety of purposes relevant to the kinds of virtual hardware which constitute the virtual machine. The definition of the virtual machine is kept at the level of constructable, conventional,

S_0 In/Out bit, 1 indicates the block is in core and mapping is allowed, 0 forces a mapping device trap.

S_1 Locked In bit, 0 indicated the block is locked in and cannot be referenced, 1 allows references.

S_2 Read-Only bit, 1 indicates only fetch-type references are allowed, 0 allows either read or write references.

S_3 Reference bit, this bit is set to 1 by the hardware whenever a successful map is made using this block, the bit is reset by program.

S_4 Active bit, this bit is set to 1 by the hardware whenever a store type reference is made to this block, this bit is reset by program.

S_5 I/E cycle bit, this bit is set to 1 by the hardware whenever I-time (instruction fetch) references are made to this block, bit is reset by program.

S_6 Conditional Protection bit, 1 indicates only instructions executed from a "privileged" block are allowed to make stored type references, 0 indicates no conditional protection.

S_7 Privileged bit, instructions executed from this block are privileged, to store, see S_6.

S_8 Transfer Protection, 1 indicates only transfers of control into this block will be allowed if they come from "transfer privileged" blocks.

S_9 Transfer Privileged bit, instructions executed from this block are allowed to transfer to transfer protected blocks, see S_8.

Figure 3

44X operating system

The 44X Operating System contains a collection of program processors similiar to those found in most operating systems. Specifically it includes:

FORTRAN IV compiler
SYMBOLIC ASSEMBLY PROGRAM
BINARY LOADER
FILE MAINTENANCE SUB-SYSTEM
COMMAND LANGUAGE SUB-SYSTEM
DEBUGGING AND CONTROL
LANGUAGE SUB-SYSTEMS

In most respects these programs are conventional processors found in most computing systems. They

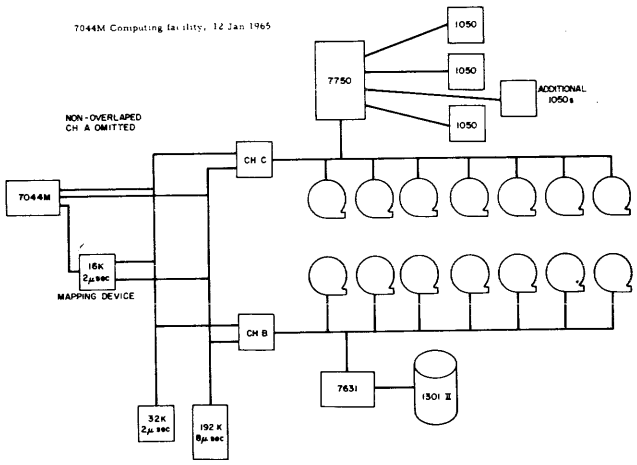


Figure 4

are unique only in their storage management. All of these sub-systems are permanently resident in the virtual 44X execution store. The collection of programs making up the 44X Operating System occupy less than 65,536 words of the 2 million word exectuion store of the 44X. Consequently the core resident system does not significantly restrict the execution store available to the problem programmer. Because of the generous size of the execution store, problems of intermediate storage management have disappeared from these programs. All of the 44X Operating System has been made Read-Only code by separating the modifiable instructions from the straight line code and executing them using the XEC instruction. Note that while this allows the same copy of the 44X Operating System to be used by many 44X's simultaneously, it does not allow recursion within a single 44X. This is not re-entrant code.

MOS, the M44 modular operating system

The Modular Operating System (MOS) is the program which honors the 44X machine definition on the hardware machine, the M44. MOS is a modular operating system so that many of its design parameters can be easily modified to facilitate system experimentation.

MOS is basically a set of interrupt handling routines whose logical inter-connections are created by the data upon which they operate. These data are for the most part maintained in a series of queues, the most significant of which is the joblist, or CPU queue. This is a list of the States Of Machine (SOM's) of the virtual machines. An SOM contains all essential information about the state of a virtual machine, such as the contents of the AC, MQ, XR's, etc. Each SOM also contains an indicator of the virtual machine's availability for processing, i.e. whether or not it can

use the CPU. The joblist is serviced by the dispatcher routine (DISP) which is executed whenever there is any change of status in the joblist. The dispatcher searches the list, starting at the top, for the first available SOM and then turns the M44 CPU over to that job. Thus there is a strict priority inherent in the ordering of the joblist. The joblist is divided into three sections: the highest priority is the operating system service routines, next the terminal oriented 44X's, and finally non-terminal oriented 44X's (sometimes referred to as background jobs). In addition to these sections of the joblist, there is at the bottom of the list an operating system routine called IDLE which is always ready to run and keeps track of unused CPU time. Ordering of jobs within the joblist may easily be changed dynamically in response to changing requirements. One of the mechanisms which changes the joblist ordering is the time-sharing algorithm.

Time-sharing is controlled by the interval timer subsection of MOS by the following procedure. A timer interrupt transfers control to the interval timer subsection of MOS, after the state of machine has been saved in the SOM entry of the job which was interrupted. The interval timer subsection then changes the priorities of one or more jobs in the joblist, initializes the interval timer for some interval, and then transfers control to the dispatcher routine. The method by which the priorities are changed, and the time interval chosen is not restricted, and therefore the time-sharing algorithm is quite easily changed. Note that there is no direct connection between the operation of the time-sharing algorithm and the storage allocation.

The mapping device interrupt handling routine provides dynamic allocation of the real core store to meet the demands of the virtual machines. The 44X demands are signalled by mapping traps. The mapping device routine (OLSR) handles these storage requests in the following manner. After saving the state of the machine in the interrupted 44X's SOM, control is given to the mapping device routine. The trap is an indication that a virtual machine is attempting to reference a logical address in a page not currently in the M44 core store. Since the mapping device operates upon logical pages not individual words, OLSR must effect the input of the logical page containing the word being referenced if the 44X is to continue processing. OLSR has access to a directory of the core image location (disk address) of all logical pages belonging to 44X's. This provides the location of the page to be transferred into the M44 core. The choice of a core input location generally involves the overwriting of a page in core belonging to some 44X (it may or may not belong to the 44X for whom the new page is being input). The proper choice of the page to be overwritten is critical to the efficiency of the M44/44X system, since poor choices significantly increase the number of disk to core transfers. This choice is accomplished by the replacement algorithm (REPL) which is quite easily changed and it is one of the primary areas of experimentation within the M44/44X system. The page chosen for replacement may or may not require saving before overwriting, depending upon whether it has been changed since its last input. This is indicated by the altered status bit of its mapping memory word. If required, OLSR arranges fot its output before it is overwritten. In any case the interrupted 44X cannot continue processing until the page referenced is in the M44 core. Therefore the SOM of the interrupted 44X is made unavailable (not ready to use the CPU) until the exchange of pages is completed and the mapping device is updated. This automatically inhibits the DISP routine from giving control to a 44X which cannot be processed.

In general two methods of 44X creation are used: those 44X's created to process on-line users requests and those created to process standard stack jobs. Note that the resultant 44X's created are not in any way incompatible and can be distinguished only by the presence or absence of an on-line terminal. The method by which the 44X's are created differs although the resulting 44X may not. The creation of a 44X to handle the processing of an on-line users request is started whenever the M44 receives an attention interrupt from a terminal which is not assigned to an existing 44X. In response to the interrupt a minimal 44X is created, i.e., 2 million words of core store including the read/only systems programs and the on-line user terminal. An SOM is created and added at an appropriate priority level in the joblist. The initial SOM instruction counter contents will contain the transfer address of the 44X systems program used to deal with on-line users. Thus when the 44X created is given control by DISP the 44X system will communicate with the on-line user and further interaction between the 44X and the on-line user will be under the control of the 44X system program designed to deal with on-line users. In the event that the on-line user requests the use of additional I/O units (tape, disk, another terminal, etc.) the 44X system program can request the assignment of additional I/O units for this 44X using a well-defined set of 44X instructions. On the current system, MOS will only honor requests of this type when they are executed from the 44X locations known to contain the 44X system programs. However, this convention is not an absolute requirement of this system organization, but without

it 44X program bugs could demand excessive numbers of I/O units and therefore slow down the operation of the system. It is possible that requests for additional 44X components for a particular 44X cannot be met immediately. In this event control is returned to the 44X system program at a fixed location, so that the on-line users program may give the user the option of waiting for their availability, proceeding with some other jobs, or discontinuing his run entirely.

44X's are created to process *stacked* jobs whenever MOS is informed by the machine operator that background work is ready to be processed. It is expected that as the system develops the operator will provide a backlog of work for the system and MOS will create 44X's to process this work when its load indicates that extra processing capacity is available.

At the time of the creation of a 44X the logical address space to be used by the 44X must be given storage area somehwere in the computing facility. Due to the nature of its use (essentially random, by pages) the only practical storage device in this system is the 1301 disk. However, the assignment of storage space for a 44X's complete logical addressing capability is not practical. The 1301 disk has a capacity of approximately 9 million 36 bit words. If the complete 44X virtual memory were kept on the disk, only four 44X's could be accommodated at any time. This is too few 44X's. Furthermore in any on-line multi-access system it is necessary to have storage for long time storage of user's private files, and this demand must also be accommodated on the 1301 in this system. Therefore the complete 44X virtual memory (2,097,152 words) is not automatically assigned storage space on the 1301. All 44X's are assigned storage space on the disk for the highest 65,536 addresses. In fact the assigned disk tracks are the same for all 44Xs; the read-only 44X system. Another 16,384 words of private storage is assigned for the lowest 16,384 addresses at the time of the 44X creation. This may be thought of as the minimum 44X virtual store claim. At any time during the processing of the 44X more virtual storage space may be claimed. It is possible that a request for more virtual address space cannot be honored, because insufficient disk space is available. In this event the 44X user is given the option of continuing with the space he already has, or halting processing for resumption at some future time when his requests can be met. If this happens too frequently it is an indication that the system is out of balance and more disk (or drum) storage is required.

44X's are destroyed by MOS in response to a 44X instruction requesting self-destruction (POOF). Since this operation will not leave any trace of the 44X which executes it, MOS will only honor this instruction when it is given from a fixed location known to be part of the 44X system control program.

It has previously been stated that certain modules of MOS are replaceable and are changed for experimental purposes. The following modules are debugged and have been used with MOS for measurement purposes.

Time-sharing algorithm 1

The first time-sharing algorithm implemented as an MOS module is a very simple one, but it has also proven effective for this type of system organization. This time-sharing algorithm shares the priority implied by the ordering of the CPU queue on a time basis. It does this by reordering the CPU queue at the end of each time quantum. Since the response time of the 44X's with terminals is of importance, only the middle section of the CPU queue is reordered. This section is reordered by removing the top entry and reentering it at the bottom of this section of the queue. This action does not necessarily affect the allocation of the CPU since higher priority jobs (MOS service jobs) may require the CPU both before and after the CPU queue reordering. In contrast to most other time-sharing systems this action has no direct effect upon the real core allocation. Real core is allocated only on demand and in a manner defined by the current replacement algorithm module of MOS. The time quantum which defines the frequency of joblist reordering is a constant in this algorithm.

FIFO replacement algorithm

The initial algorithm implemented to allocate the real pages in the M44 was a FIFO (First In, First Out) algorithm. This algorithm was chosen for no other reason than its ultimate simplicity. It provides a benchmark to measure subsequent replacement algorithms against, and it increased the speed with which the initial version of MOS could be debugged. Its effect upon system performance does not recommend it. The FIFO replacement algorithm chooses the real core page to be overwritten by finding the page which has been in the real core for the longest period of time. In practice this means it picks pages to be replaced in a round-robin fashion among the pages allocated to virtual machine pages. This algorithm is extremely crude and has been known to make the worst possible choice of pages for replacement. It has overwritten the page containing the instruction causing a mapping trap, in order to bring its data into the real memory. Of course when the data page is in real core it is now necessary to replace another page to bring the instruction back to use it! Fortunately it cannot get into a loop of this type.

AR replacement algorithm

On the basis of extensive simulator studies of 7094 programs (reported in the IBM Systems Journal, Vol 5, No. 2, "A Study of Replacement Algorithms for a Virtual Storage Computer," by L. A. Belady) the AR replacement algorithm was the most promising page replacement algorithm which was studied. This algorithm required some additional hardware on the M44 and consequently was not implemented immediately. The "AR box" hardware contains a bit for every page in the real core which is used for paging. Each bit is set by the hardware whenever a reference is made to its corresponding real page. The complete set of bits is scanned by the hardware after every memory reference; if all of the bits are set they are all reset. This mechanism guarantees that there is always at least one unset reference bit. The AR replacement algorithm chooses its replacement page from the set of pages which have not been referenced since the last reset. Hopefully the lack of reference in the recent past indicates that these pages will not be required in the near future.

Biased replacement algorithm

In addition to the replacement algorithms which have been described, tests have been made using a biased version of the FIFO replacement algorithm. The biased version differs from the non-biased replacement algorithm by the addition of a restraint. A favored 44X is chosen and the replacement algorithm is biased in favor of the choosen 44X by rejecting for replacement any pages belonging to the favored 44X. The designation of the 44X for which the system is biased is changed on a regular basis. The biased FIFO algorithm picks a new 44X after every second cycle of complete search the real memory. This implies that the rotation of memory preference is a function of the page replacement rate and the percentage of pages belonging to the favored 44X.

M44/44X system performance measurements

Measurement and evaluation tests have been made for two distinct purposes: comparison of virtual machine organization and more conventional system organizations; and to evaluate the effect of various parameters on the performance of virtual machines. The effect of various virtual machine parameters will be discussed first.

To simplify the comparison of test results all testing was done using only the 8μsec memory. Since all virtual machine execution is done in the mapping mode, and the 2μsec mapping cycle is not overlapped, the effective cycle time of the 44X is 10μsec. Significant parameters for the interpretation of all tests results are: type of work load processed; page size used; time-sharing quantum, if the time-sharing algorithm was used; real core size; and the degree of multiprogramming i.e. the number of virtual machines processed concurrently. For all the test runs which will be discussed these parameters will be specified, but it is very important to keep in mind that the results apply only for the particular combination of parameters used. Unless otherwise specified all loads are processed using tape for Sysin and Sysout.

Effect of page size

The virtual machine organization separates the logical memory addressed by problem programs into pages of a fixed size for the purpose of storage allocation. This implies that when any particular word of the virtual memory is required for execution, a full page of the logical memory must be present in the real execution store. Because of the regularity of most programs some of the words in the page will probably be required in the immediate future for execution; however not all of the words will be required. Those words which must be in the execution store due to the arbitrary break-up of programs into pages, but are not required for execution, represent wastage of real core. (Note that non paged system organizations do not avoid similiar wastage of execution store, since storage claims generally exceed those necessary for execution.) The amount of core required for this type of wastage depends upon two factors, the organization of the program and the size of the pages being used. Figure 5 illustrates the effect of different page sizes upon the amount of execution store required to fully accommodate the needs of two different job loads. For the load marked A-taped (10 small FORTRAN compilations and loads), the required core goes from 20,992 words to 73,728 words. With 8,192 word pages it is clear that at least 62,736 words of execution store is wasted since the same program never references more than 20,992 words when 256 word pages are used. Some of the core required even with 256 word pages is unused, but how much is not clear from the data at hand. That the wastage caused by large page sizes is dependent upon the load is clear from the core requirements of the job market FTT (FORTRAN compilations, and executions, used to debug the 44X FORTRAN compiler). The increase in core required for this job is a small percent of the total requirements as the page size is increased.

The previous curves represented the total core required for the complete processing of some jobs. However because of the paging mechanism it is not necessary to devote this much core to a job at any one

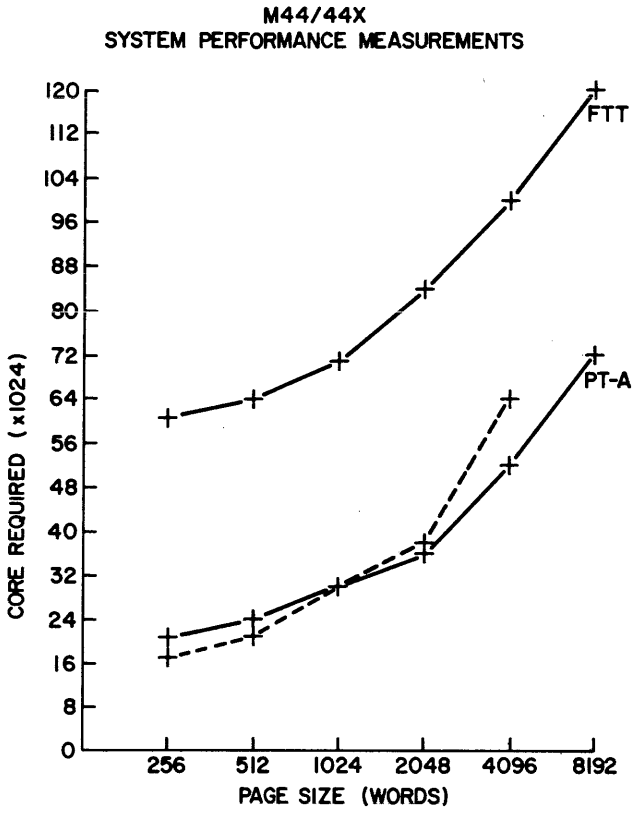


Figure 5

moment during its processing. Paging provides an automatic overlay mechanism when less than complete program requirements are available. Figure 6 shows the effect on thruput of reductions in the amount of core devoted to a particular job's requirements. As the amount of core is reduced, more paging activity is necessary and performance suffers. The performance degradation is slight up to the point when not enough pages are available in the M44 to fully accommodate the basic segments of the job load. At that point performance suffers drastically. The degradation in performance small job (PT-A) is more sudden than that for the job which requires more execution store to meet its complete requirements. This occurs because the size of core necessary to completely accommodate the major loops of the programs does not vary nearly as much as their total execution store requirements. Large programs tend to consist of more segments, not just bigger segments. It is worth mentioning that even at the worst performance point shown for the larger job (FTT), which is with 8,192 words of real core, its processing time is no more than that required to process this job on the 7094 II under IBSYS. The change in storage management technique has gained quite a bit in this case.

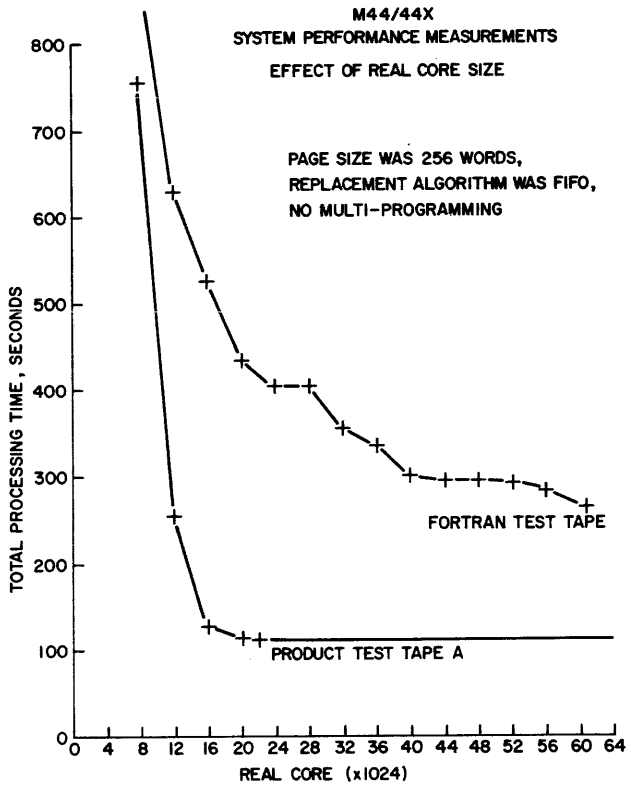


Figure 6

Effect of multi-programming

Performance of the M44/44X system may or may not be improved by multi-programmed operation depending upon a number of factors. Significant among these factors are: the amount of real core available to the system, the number of autonomous devices in the system (i.e. channels and processing units), and the meshing of the requirements of programs being multiprogrammed. Figure 7 illustrates the effect of the level of multi-programming (i.e. the number of virtual machines processed concurrently) upon CPU efficiency. Efficiency increases with increasing load up to the point where competition for pages in the real core between virtual machines creates excessive amounts of page turning activity. With excessive page turning performance suffers drastically, since the typical state of the virtual machines becomes one of "page wait." This property has several important implications for system design. The operating system should continuously monitor its operation, and automatically reduce the level of multi-programming, by setting work aside, if it detects an overload condition. Second, the system must have adequate execution store if the full benefits of multi-programming are to be realized.

Effects of time-sharing

The time-sharing algorithm module of MOS was primarily intended to insure an adequate response time

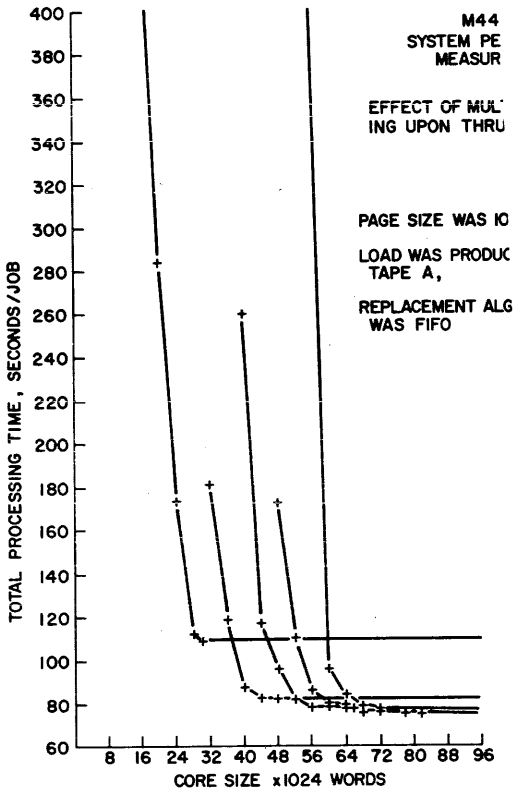


Figure 7

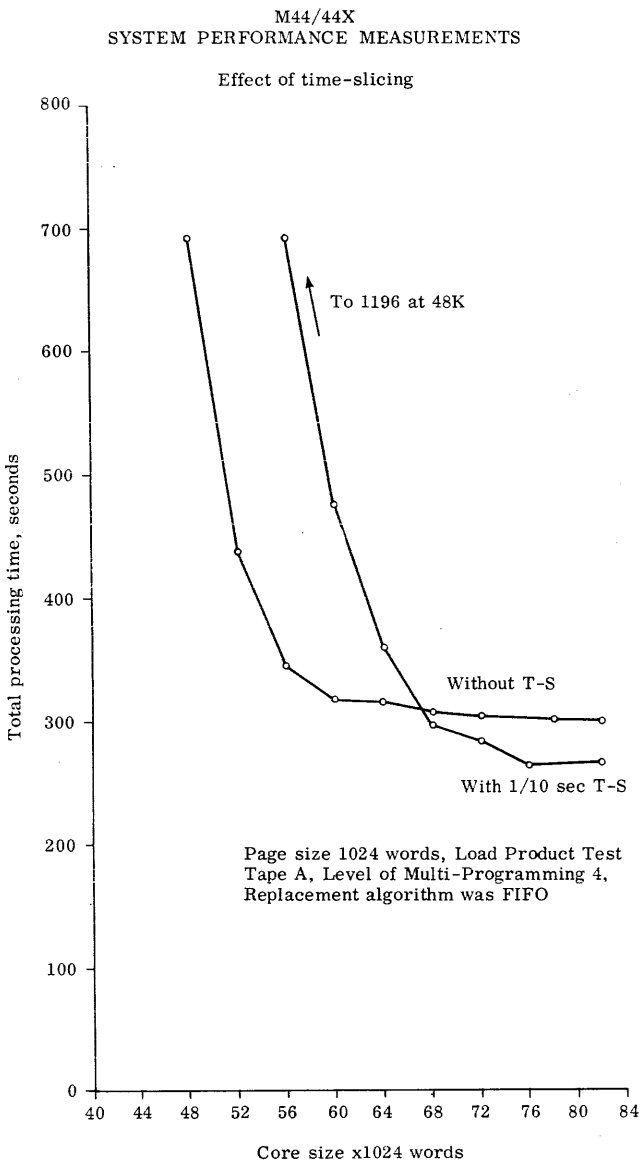


Figure 8

on those 44X's with terminals attached. However, provision was made to allow the time-sharing of non-terminal 44X's when turnaround time was of great importance. Experiments using this capability demonstrated that time-sharing often has significant advantages even when turnaround and response times are not important. Figure 8 shows the total time required to process the same load with various amounts of real core, both with and without the time-sharing algorithm in operation. In this example with more 64K of memory the total processing time is less with time-sharing than without. When less than 64K of memory is available the system would function more efficiently with time-sharing and a reduced level of multi-programming although this is not illustrated by Figure 8. The explanation for this behavior might be summed up by saying "the more you stir the pot, the more mixed up it becomes." In more technical terms what appears to be happening is a demonstration of the fact that a program which has run on the CPU for a time period t, without any I/O request, has a lower probability of requiring I/O than a program which has not used the CPU. Therefore frequent switching of programs tends to insure that the overall balance between demands for CPU and channels is better. Some testing has been done with time-slice quanta other than 1/10th second. The effect of varying the slice in the range 1/60th to 1 second is negligible. The importance of the time-sharing algorithm on a system of this type is not great, since it has no direct effect upon the storage allocation.

Comparison of the M44/44X system with other computing systems is inevitably a comparison of the total systems including both hardware and software packages. Therefore comparisons of this type have a somewhat limited significance. Unfortunately there does not exist any satisfactory alternative to this type of comparison. One possible way to evaluate systems is to compare the time, or cost, required to perform some "typical" job. This we have done, but it must be emphasized that in real life no particular job is "typical," and therefore the results are biased by the jobs chosen.

Comparisons have been between the 7094 and the M44/44X using three distinct test jobs: Product Test Tape A,B, and C. Product Test tape A contains 10 relatively small (about 200 statements each) FORTRAN IV source language main programs each with several sub-programs. Product Test tape B consists of 4 FORTRAN source language programs for compilation and execution, with all four of the object programs very heavily oriented toward numerical calculations with no problem program I/O at all. Product Test tape C contains 6 FORTRAN programs, each of which performs conversion for output, and writing of 10,000 records on private tapes. In order to properly evaluate the comparative timings of the 7094 II and the M44/44X system it is necessary to bear in mind that the memory cycle times are 1.4 μsec and 10μsec respectively, and the tape speeds are 90KC and 60KC respectively.

Test tape A requires 6 minutes and 24 seconds to compile all 10 programs on the 7094 II, with no loading or linking of the object decks. On the M44/44X system, utilizing multi-programming at the level of 4, with 1/10 second time-slicing, all problems on Test tape A can be compiled, loaded, and linked in 50 seconds. This is clearly a load better suited to the M44/44X system organization. The most significant gains of the M44/44X system accrue from the in core compilation, the elimination of separate loadings of the compiler for each compilation, and multi-programming. All three of these are directly related to storage management.

Product Test tape B requires 5 minutes to compile and execute on the 7094 II and 27 minutes 43 seconds to compile and execute on the M44/44X system, with no multi-programming. It is interesting to note that the ratio of completion times is almost exactly the ratio of the memory speeds of the systems. No multi-programming was done for this comparison on the M44/44X since nothing can be gained by multi-programming several jobs all of which require only use of the CPU. This type of job may be advantageously multi-programmed with more I/O oriented jobs. When this is done, the compariosn between systems can no longer be made as any assignment of increased speed to particular jobs of the multi-programmed mix is purely arbitrary.

Product Test Tape tape C requires 15 minutes to compile and execute on the 7094 II and 17 and ½ minutes on the M44/44X; when the M44/44X is multi-programmed to the level of 2 (the number of overlapped tape channels on the M44). Better thru-put can be achieved from the M44/44X system by combining the B and C tapes. For example, two C tapes and one B tape run rogether require 57 minutes and 31 seconds to complete on the M44/44X. With all three jobs started simultaneously, one C tape finished after 30 minutes, the other C tape after 37 minutes and the B tape after 57 minutes and 31 seconds. Since the last 20 minutes were not multi-programmed, further gains could be expected from further mixing of the load.

On-line time-sharing experience

In addition to the controlled experiments run with background jobs considerable experience has been gained while the M44/44X system was providing computing service to on-line users. The results of this experience are not amenable to the analysis which can be given to the results of controlled experiments. Changes in system parameters are often not as important in changing the system characteristics as are the changes in the behavior and composition of the individual terminal users. However the results of monitoring on-line usage do have value. Some of the data monitored have shown remarkable consistency in the face of both system changes and user changes. Correlations between properties of system behavior are also of interest for system analysis. Finally some feeling for the behavior of the users and the varying responsiveness of the system can be of value even though it can not be precisely quantified. Because the system was designed from the start as an experiment in system design many statistics are gathered by the system. At the end of every system up-time period these statistics are reduced to readable form and saved.

Idle time significance

The measure of performance has been processing time, i.e., the time required for the system to completely process the job load. For simple types of measurement this is a satisfactory measure, but it is inadequate for comparisons of performance on different job loads. Some jobs are inherently short jobs and others are long. A more satisfactory measure is the efficiency with which the hardware components are used. A somewhat indirect, but effective, measure of CPU usage is the percent of time spent in the MOS "idle loop." The "idle loop" time is wasted CPU time since it is precisely that time when the system has no meaningful function to perform on the CPU for either problem program or control program requirements. This occurs when all virtual machines are in wait status, awaiting completion of I/O. The I/O may be either problem program I/O or paging I/O. The "idle loop" times should not be used to make direct comparisons with non-paging systems, since idle times may be attributable to either paging induced

activity or to problem program I/O which is non-overlapped.

The latter contribution to the idle times is part of the inherent nature of the job being processed rather than an attribute of paging systems. In addition some of the paging I/O (ex. initial program load) shows up on other systems although not in the form of paging. The time required for the housekeeping activities of the control program (MOS) was not measured since it is difficult to distinguish between paging induced housekeeping, and control program execution of necessary problem program functions. With these reservations in mind the "idle loop" percentages can be viewed as a comparative measure of the efficiency of various configurations of the M44/44X system.

Figure 9 shows the page replacement rate and the percentage of CPU utilization (non-idle time) on a minute by minute basis during a prime-shift time-sharing session. The figures in the left-hand column are the actual number of replacements during a minute, times the page size divided by 256. This method of normalization allows comparisons of paging rates even though different page sizes were used. The relationship between the paging rate the CPU utilization, while not completely regular is apparent. The CPU utilization appears to be inversely proportional to the paging rate. However when the paging rate goes down this clear relationship disappears. Figure 10

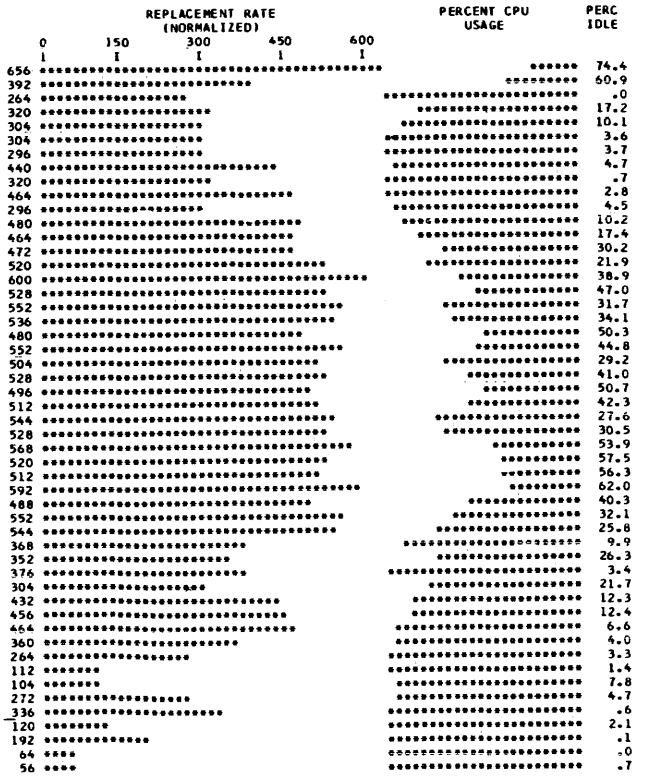


Figure 9

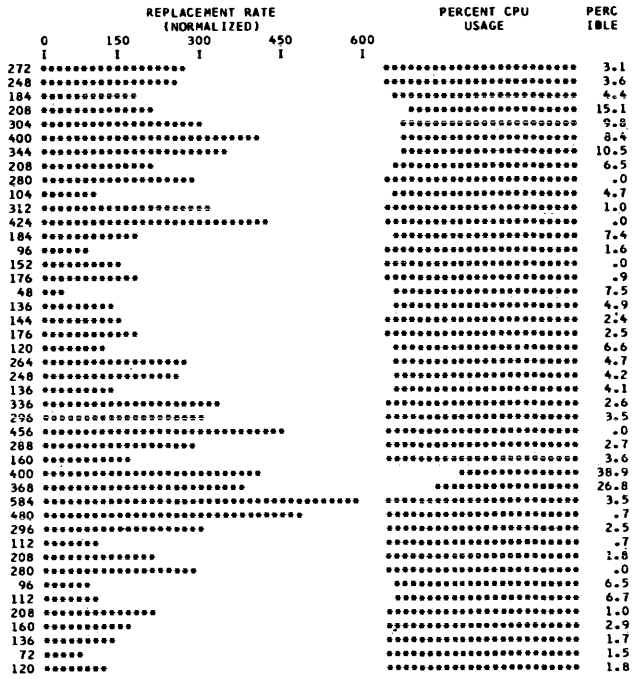


Figure 10

represents the next 44 minutes of the same session as shown in Figure 9. In Figure 10 the CPU utilization appears to be independent of the paging rate, although the page rate varys rather widely. During the time-sharing operation of the M44/44X system Figure 10 represents a more typical state of the system than Figure 9. The importance of these two curves is that under overload conditions the paging rate can get out of hand and reduce the overall performance of the system dramatically. However a properly designed system can protect itself against such performance degradation. It is necessary that the system monitor its own performance and when an overload is detected it can defer some of its load to a period of less demand.

The wide variations in the paging rate, even when monitored on a minute by minute basis, illustrates the difficulty of using the average paging rate as a measure of system performance. In addition to the wide variation in paging rate there is a difference in the load upon I/O channels due to paging, depending upon the type of pages being brought into the real core. The page being replaced may or may not require saving depending upon whether it has been modified since its last input from a secondary storage device. Similiarly a page being called for may or may not require input from a secondary storage device depending upon whether this logical page was ever used by the virtual machine before. A page which has never been referenced by the virtual machine cannot have data meaningful to the 44X in it, in which case any page in

the real core will do. Because of these two possibilities the amount of I/O required to complete a page replacement is of interest when deciding upon secondary storage facilities for a paging system. During monitored time-sharing with the M44/44X system the number of pages transmitted to and from the secondary storage device has varied from 1.1 to 1.5 page transmissions per replacement. The higher figures in general representing sessions during which the average page replacement rate was higher than normal.

The section of the 44X virtual store which contains the 44X system programs is read-only and consequently a single copy in the real memory can be shared by all of the 44X currently using the system. The implementation of MOS provides for such sharing of the virtual system pages. A count is maintained which reflects the average number of virtual machines which have referenced each system page during its lifetime in the real core. During time-sharing sessions when up to 16 users are allowed on the system this shared ratio has varied from 3.5 to 5.0 when the FIFO replacement algorithm is being used. On the basis of this data it is clear that the ability to share frequently used system programs significantly improves the utilization of the execution store of a paging system.

SUMMARY

Experience to date with the M44/44X system has demonstrated the feasibility of the concept of dynamic relocation using paging hardware. Some improvements in the implemented system have become obvious, but even the initial implementation appears to have significant advantages for some types of processing. The controlled system experimentation done on this system represents a useful first step in advancing system design from its present state as an art to an engineering science. Large amounts of data have been, and are being gathered by this system relevant to paging system behavior. The significance of much of these data is not yet clear. Much remains to be done.

THOR–a display based time sharing system*

by JOHN McCARTHY, DOW BRIAN,
GARY FELDMAN and JOHN ALLEN
Stanford University
Stanford, California

INTRODUCTION

THOR is a time sharing system for the PDP-1 computer with the capacity to run twenty user programs. The system has twenty-eight user consoles, twelve of which are combination keyboard and cathode ray tube display consoles. THOR is designed to capitalize on the display's ability to present large quantities of information quickly and to mitigate the fact that hard copy is not available at display consoles. The other sixteen consoles are Model 33 teletypes with the attendant slow presentation of information and the availability of hard copy.* Because there are more consoles than user programs available, a user program may use more than one console.

THOR was designed to serve a number of purposes:

1. To control the computer-based teaching laboratory. In this application the displays and several other devices including six film chip projectors, a system for presenting audio messages, and teletypes located in schools, are used as teaching consoles. The teaching applications are run as standard time-sharing activities. This has proved quite important to ordinary users and to the teaching laboratory personnel since they require very large amounts of console time for debugging new programs and editing text material.

2. Text editing. Facilities are provided for keeping texts on disk files, creating new files and editing old ones. The texts may represent programs in a variety of languages, teaching material, or any other information organized in pages of lines of characters. Most of the console usage is spent in text editing. The display based text editor has proved itself more convenient and faster than other systems. The lack of immediate hard copy has not proven a serious disadvantage. (Hard copy, when desired, may be obtained by teletype or line printer.)

3. General purpose programming for a small computer. A variety of systems are available including assembly language, an algebraic compiler, an interpreter, a variant of LISP 1.5, and a system for manipulating and displaying functions represented by graphs.

4. To provide time shared access to the IBM 7090 from the display consoles. This has been available to a limited extent at various times. Unfortunately, the 7090 batch processing system has required repeated modifications to give new facilities so the time sharing work has suffered.

The following are the main results of the project:

1. Displays provide a significant improvement over teletypes as time shared consoles. Users decisively prefer them. The large (114) character set and seven character sizes proved valuable.

2. Powerful interactive systems for text editing, on-line debugging and system control have been developed. A flexible system of instructions has been developed which allows the user to design his own interactive systems.

3. It has proved practical to combine a general purpose time-sharing activity with the teaching machine project, a major special use that requires high reliability.

4. Insight has been gained in understanding the nature of tradeoffs in a time sharing system between efficiency, core space, generality, and flexibility.

Details of these matters are given in the following sections.

Hardware

The computer in this system is a Digital Equip-

*Stephen Russell, Brian Tolliver, David Poole, and Paul Stygar also contributed to the work and the paper.

**Display consoles have proved so superior to teletypes that the latter have been retained only as input/output devices in the latest version of the system.

ment Corporation PDP-1, a single-address, 18-bit binary machine.* The central processor has 32 instructions and can address 2^{12} words directly and 2^{16} words indirectly. It has a well developed interrupt system with 16 separate interrupt channels organized in a priority scheme to prevent a lower channel from interrupting a higher priority channel. The input/output connections are easy to modify and inexpensive to extend. This machine lacks an index register and floating point instructions.

There is a restricted mode of operation normally imposed by the system on user programs. In this mode, all attempts by a user program to reference outside of an assigned core area, do input/output, or stop the machine cause interrupts. This lets the system confine the user program space and interpret his input/output commands.

The core memory is attached to two separate controls. The selection of control is made on the high order bit of the address. Each control has connections for four independent devices: the drum, the disk data channel, the display data channel, and the central processor, in descending priority. When idle, each memory control gives its next memory cycle to the highest priority device requesting a cycle from that control. Thus, two of the devices in the system may be getting data from memory at full memory speed simultaneously. For example, the drum may be swapping users from the higher memory while the central processor and the display processor share the lower memory uninterrupted.

Basic to the time sharing system is a very high speed drum whose basic operation is a swap.† In this operation the contents of 2^{12} locations are transferred from core to a drum track, and simultaneously the same core locations are loaded from a different track. This swap takes 33 milliseconds regardless of drum position.

Twelve display consoles serve as the primary user stations. These consoles are capable of displaying 114 different alphanumeric characters, as well as arbitrary line segments (called vectors) and randomly positioned points.

The characters are generated automatically by the display controlled from six-bit binary codes; characters may be displayed in any of seven program selectable sizes. The time required to generate and display one character is only five microseconds. Line segments, or vectors, require five to fifteen microseconds to display. Both characters and vectors may be displayed at any of three intensities. A vector is represented by an 18-bit computer word, which specifies the horizontal and vertical components of that vector. The origin of the vector is ordinarily taken to be the end point of the previously displayed vector. Thus a displayed figure consisting of many line segments may be moved to a different position on the screen by changing only the origin of the first vector in the figure. This ability has proved very useful for programs displaying moving pointers.

The display consoles are all driven and controlled by a single central logic unit. This greatly reduces the total cost of the system, as each console unit can be comparatively simple in design. The information to be displayed on a given console is organized by the program into a table in computer memory. The information in this table will consist of a mixture of control words, six-bit codes specifying characters, and words describing vectors. Approximately once every 30 milliseconds the monitor program starts a data channel which transmits the contents of all the display tables to the display controller, one word at a time. The central logic unit, guided by the control words, displays the information in each table on the proper console. Because the display information is stored in computer memory, small parts of the display can be modified quickly and simply, utilizing the instructions of the main computer. This ability is especially useful in text editing and in other programs requiring rapid visual interaction.

Each display is the size of a small refrigerator with a 16-inch cathode ray tube, and a keyboard mounted at desk height. (See Figure 1 "A Display Console".) The keyboard has 64 keys, with the numbers and letters in a standard typewriter arrangement, with the additional keys at the right side. In addition, there are two control buttons to augment the character

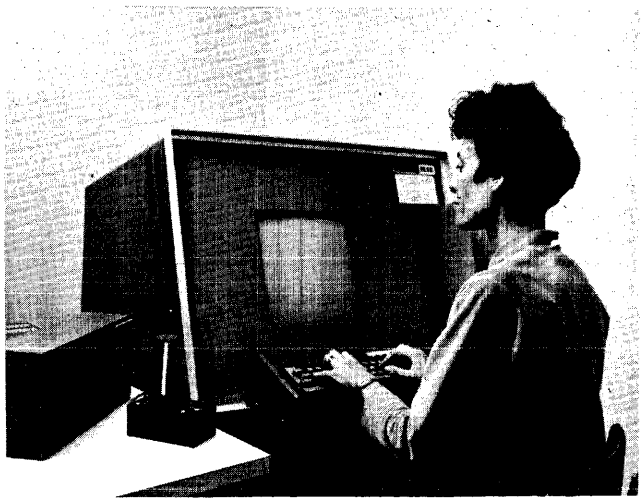


Figure 1—A display console

*The basic design of this computer was done by the late Benjamin Gurley.

†The idea of the swapping drum for time-sharing is due to E. Fredkin.

code. The keyboards on the displays are logically separated from the rest of the display system, and have a separate interface to the central processor.

When a key is struck an interrupt is sent to the computer and the keyboard locks until the computer reads the keyboard scanner. The computer interrupt program reads the console number, the 6-bit code corresponding to the key, and two bits representing the state of the two control buttons.

Other input/output devices include sixteen teletypes, a data channel to an IBM 7090, an IBM 1301 disk file of 50×10^6 characters, shared with the IBM 7090, and an analog to digital converter. The teaching laboratory consists of six installations each containing a display console, an experimental IBM film-chip projector, and a Westinghouse audio station. The film-chip projector contains 256 microfilm frames which may be projected one or two at a time on an $10'' \times 13''$ screen. There are eight masks underneath the screen which may mask off parts of the image. The projectors can detect a light pen, a feature which allows user interaction.

The audio system is a Westinghouse Prodac-50 computer which controls twelve random access audio tape drives, each of which has 1024 two second messages.

Both the film-chip projector and audio units have maximum access times of two seconds or less.

System configuration

The user is a person who is running a computer program within the THOR time sharing environment. He has a charge number assigned to him which determine his identity for THOR. This number identifies his disk files on a permanent basis; and, when he is using the system, it identifies his console, his individual drum track for storing binary computer programs, and whatever other facilities he may be using at any given time. While the user is logged into the system, he owns at least one console and one drum track. The console consists of a keyboard which allows the user to type information to his user programs and to THOR, and an output device, either CRT display or teletype printer. This output device provides a slate for the user programs and THOR to communicate with the user. When the program is actually being run by THOR it is swapped into PDP-1 core from the user's drum track.

The ordinary user program may occupy 4K of core; however one may request up to 8K additional core. Up to twenty programs may be run by a regular process of bringing a program into core from the drum, allowing it to execute for a short time, marking the state in which its execution is stopped, returning it to the drum and picking up the next user program. User programs are serviced regularly in this fashion on a round robin basis. After a user program has been executed, it is placed last in the queue of user programs waiting to run. Each program in turn is allowed to run for one quantum of time, 64 milliseconds, and then exchanged for the next program. If only one program is in a condition to run it is allowed to run without interruption. The amount of time that the system takes to exchange programs is 33 milliseconds. This swap time is the major source of overhead. However, the swapping time is used to handle system functions and I-O buffering, so it is not entirely wasted.

There are two ways a user can place a binary user program on his drum track. He may prepare an octal program at a console using any of the debugging program such as RAID or the system's octal debugging feature, or he may load a binary image of a program from a previously prepared disk file. Binary files prepared by the assembler and compiler are in the proper format for loading onto a drum track. In addition the system provides a means of saving binary core images from a drum track onto a disk file.

FILES

The main storage device for the system is the IBM 1301 disk. The disk is divided into logical areas called reserved files. Each file is referenced through a unique number and a programmer assigned name. Facilities are provided for creating new files, extending, contacting and destroying old files. The disk file may contain textual information, binary core images, or scratch data in any format. The user may protect his file against unauthorized access.

The system maintains three "internal file numbers" for each user. The user may associate a reserved file with each internal file number. All disk commands then operate in terms of the internal numbers, rather than in terms of the actual name and number of the file. Through this device a user may attach one of his files to an internal file number, then load utility or other programs which operate on his file without having to explicitly open the file for each utility.

With moderate system activity, ten to twelve users, one can expect references to disk file to take no more than 250 milliseconds.

Dsiplays under THOR

We wished to make the display consoles as easy to use as a teletype and yet allow the user access to the full generality of the displays. To this end two display buffers are associated with each of the CRT consoles. One is called the "page printer" buffer and the other is called the "free" buffer. Only one of the

two buffers may be in core at a time; the other is stored on the drum. The user program can control the visibility of these buffers by either executing an instruction which explicitly calls one of the display buffers, or by executing an instruction that implicitly calls a particular buffer.

The "page printer" is used to output characters on the displays in a standard format. When the "page printer" is being used, characters placed in the output buffer of a particular console are displayed under automatic control of the system. As new lines of text are added at the bottom of the display, old lines disappear off the top. A carriage return is automatically inserted whenever a line exceeds the width of the screen, and an * appears at the beginning of the line's continuation. The character, backspace, erases the last character displayed and moves the display pointer back one space. All system messages to the user appear on the page printer. In addition many user programs elect to use this form of output.

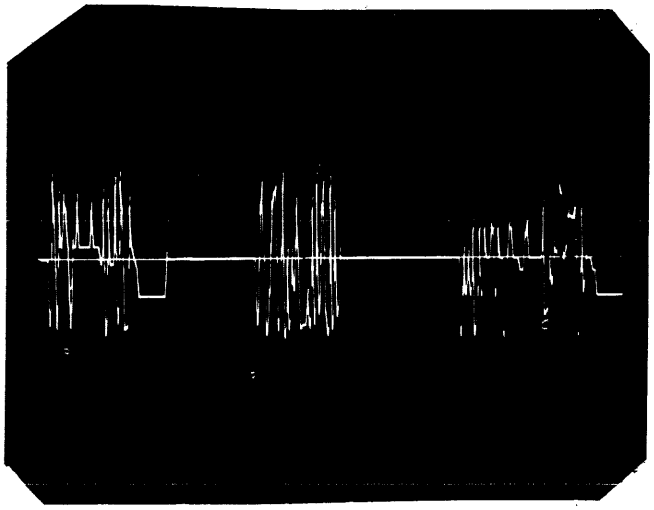


Figure 2 – Display of speech segment
This display was made with vectors deposited in the user's "free" display buffer

The "free" buffer is used for any displays which are more elaborate than the simple page printer. The user is allowed to write display information in the basic language of the display hardware. An example is found in Figure 2 "Display of a Speech Segment." User programs may deposit characters and vectors directly into the "free" buffer. Both the editor and the debugging program described below use this facility.

When the display load is light, a user program may access the buffers which are not in use. This feature allows a program to display on more than one console or to display extra information on a single console. This occasionally proved necessary for display of complex pictures made up of a large number of vectors.

Considering that not every program can make use of the extra core of extra display buffers, on the average less than half the machine is available to each user program. We felt that the wide variety of services that the system supplies and low overhead times possible with fast but voluminous system code are more valuable to the time-sharing user than a little extra space. In particular we felt it vital to keep the time the user waits for response from either his system or his program as low as possible. Wholly new ways of programming arise when response is instantaneous rather than even a few seconds delayed.

In-core section

The in-core section of the THOR system provides services which may be divided into three broad categories:

1. I-O channel routines
2. Scheduling and activation
3. Communication

The I-O channel routines act as the face of THOR turned toward the input/output devices. On an interrupt-priority basis channel routines receive information from all input devices, parcel that information out to the appropriate buffers, and inform the activation section when a user program must be activated to receive its information. Other channel routines accept output from user programs, buffer it, and send it to output devices whenever they are able to receive it.

Activation is handled by a round-robin scheduling algorithm with the exception that programs with disk requests pending are run out of turn to optimize disk usage. The task of the activation routine is to decide whether to remove the current program from core and if so to decide which program is to be run next. A user program may be in one of the following states:

1. **Running.** The user program is in execution. It continues execution until its quantum has expired or until it issues an I-O request that cannot be satisfied within the time remaining in its quantum.

2. **Active.** The user program is ready to run and will be swapped in when its turn comes. A program can become active when:
   a. an output buffer is *almost* empty (this assures continuous output of information),
   b. an input buffer is *almost* full (since it may take some time before the user actually gets to run),
   c. input requested by a user has arrived,
   d. a user determined activation condition has been satisfied (for example, a user program may arrange to be dismissed until a given time arrives),
   e. or when the user commands the system to begin execution of his program.

3. **Waiting.** In this state the program would be active except that it is waiting for the completion of some I-O request or special condition.

4. **Dormant.** The program is not being entered into the round robin. The user may be communicating with the THOR system interpreter or may be dismissed for a variety of user determined reasons.

A program may be swapped out for several reasons.

1. The quantum has expired. The program moves from the running state to the active state.

2. The program has requested the quantum be terminated. The program moves from the running state to the active state.

3. The program has filled an output buffer or has requested input and the input buffer is empty; or the program has requested a special dismissal condition. The program moves from the running state to the waiting state.

4. The program has tried the execute an illegal instruction. The program moves from the running state to the dormant state.

The in-core section provides mechanisms for communication among the users, user programs, and the in-core section. Communication is provided through the input/output transfer instructions (called iot's) which are trapped by the system instructions. When a program executes certain iot's, the system picks up locations in the user program as parameters to service routines. These routines may simulate input/output to the on-line device, control or release ownership of devices, handle character communication, or return information to the user program by filling registers within the program. Through iot's the user program can make its wants known and the in-core section can inform the user program of any variation in the time sharing environment.

Next to services provided by the iot's, character transmission is the major medium of communication. Characters are generated by users typing at keyboards and by user programs sending characters out. Characters go into input buffers to be read by user programs or to output buffers to be printed on scopes or teletype printers. A switchboard provides the possibility for setting up any useful character transmission path: any character source (keyboards and programs) may send to any character sink (input and output buffers). To insure that unwanted connections are not made, facilities are provided so that the user owning any sink may grant or deny permission for a connection into that sink.

This switchboard generality finds application in:

Duplexing—Characters typed at a keyboard can be printed on any console, display or teletype, without user program intervention.

Inter-program-communication—User programs can communicate with each other and with the system interpreter. Thus, several user programs may run as one system coordinating their separate tasks through character communication.

Inter-console-communication—Users at consoles can set up general links for conferences, teaching, monitoring, or chatting.

Multiple-consoles—User programs may receive characters from and send characters to more than one console. This allows a user program to act as a time sharing system within THOR controlling its own set of consoles. Applications include teaching machine monitors and games involving several players.

Character input is a major cause of user program swaps. However, programs vary in how promptly they must pay attention to incoming characters. One limiting case is in the preparation of a file. Here characters should be added to the input buffer when typed, and, when the input buffer gets full, the program should receive all the characters at once and transmit them to the disk. There is no need for this program to be activated each time a character is typed; it need only be activated when the input buffer gets full. On the other extreme is a program whose operation is controlled from the keyboard. Here every character typed should go directly to the program to have an immediate influence on the program's action. To save swaps, we allow each user program to specify under which conditions it should be swapped to receive character input.

In short, the in-core section of the system provides those services which must be performed immediately to allow user programs to continue running with as little delay as possible.

Service programs

There are other services which do not require such fast action. When it is the user rather than the user program who is waiting for the completion of a service, the system need only respond faster than human reaction time. It is of little importance, for example, if the user need wait an extra second to receive an error message. Whenever a service exists whose time of performance only need be faster than human response, that service is given by a user program rather than put within the in-core system. Depending on the nature of the service, the service user program may be given the privileged status of direct access to the in-core part of the system and unrestricted input/output.

There are three types of service program:

The System Interpreter—privileged
Phantoms —privileged

Utilities —not privileged

The System Interpreter acts as the external face of THOR. It accepts commands to THOR from the user typing on his keyboard or from a user program sending characters and communicates to the user by typing into his 'page printer' display buffer or onto his teletype printer. The 'call' character directs all subsequently typed characters to the system interpreter up to and including the next carriage return. This means the user can type requests to the System Interpreter "on the fly" while his user program is running. Because the System Interpreter may be receiving messages from as many as twenty-eight keyboards and twenty user programs it must be run often. Consequently it occupies its private position in the round robin as the twenty-first user being activated whenever there are characters in its input buffer.

A wide variety of services are provided by the System Interpreter:

1. The System Interpreter verifies the user's name and charge number on 'LOGIN' and provides him with a drum track to store user programs. It releases facilities owned by the user on logout.
2. Accounting is based on both the time spent sitting at a console and the time the user program has actually been running.
3. The System Interpreter parcels out such limited facilities as extra consoles, extra display buffers, extra core memory, and input/output devices. Extra consoles may be made 'slaves' so that it is impossible to call the system from them. Slave consoles are used only as character input/output devices. For equipment which several user programs may want to use together, a 'club' is formed with one user as president. Only he has the power to add or delete user programs from the membership list.
4. The System Interpreter provides commands for general file handling and maintenance.
5. The System Interpreter allows the user to save all or part of his binary user program at any time for future use and reference, and to restore it with its state unchanged.
6. The user can start, stop, and continue his user program.
7. The System Interpreter can provide the user with information about the state of his program while it is running, as well as information about the state of in-core tables concerning the user program.
8. The System Interpreter provides commands for calling the various utility programs.
9. The System Interpreter provides a primitive debugging service that allows the user to look at and modify all registers of his core image and look at all the registers of the in-core section of the system.

Phantoms provide a means of charging slow services to the running time of a user program. Phantoms are privileged user programs which run in place of a regular user program in the round robin. Thus, the time that the phantom takes to perform its service for a user program is charged to that user program without degrading the performance of the system for the other user programs. There are two phantoms:

The Error Phantom prints all the error messages for running user programs. Printing a lengthy message may require several quanta; the use of the error phantom 'punishes' the user responsible for the error and no one else.

The Iceberg Phantom handles all modifications to the reserved file directory such as the creation, destruction, lengthening and renaming of disk files. These operations require time-consuming references to file control information on the disk. The Iceberg may be brought into operation by a user program executing certain iot instructions or by the system interpreter acting in the name of a user program.

As an additional refinement, orders for the phantom programs are stacked within the in-core section so that, if a phantom completes a task for a particular user program before the end of its quantum, it may start the next task without additional swaps.

Utility programs are non-privileged user programs which may be called from the disk to perform the workhorse services of the time sharing system. They include a scope text editor, a teletype text editor, assembler, compiler desk calculator, and listers.

Text editor

Virtually all editing of symbolic programs is done on the display consoles, using the TVEDIT text editor. The editor is oriented toward the average user of the system, not just the expert programmer. The central design objective was a simple, easily remembered command structure, which would not require the user to have any knowledge of the manner in which the files are actually stored on the disk.

TVEDIT is a random-access editor interacting with the user on a character-by-character basis. Any change in the text directed by the user is immediately reflected in the display, and the appearance of the display at any time is an accurate picture of the current status of the text file and the editor. An example of displayed text is found in Figure 3 "TVEDIT text for a Demonstration Program."

Both control information and new text are typed from the keyboard. Control characters are distinguished by use of one of the special buttons on the display keyboard. This scheme is felt to be more

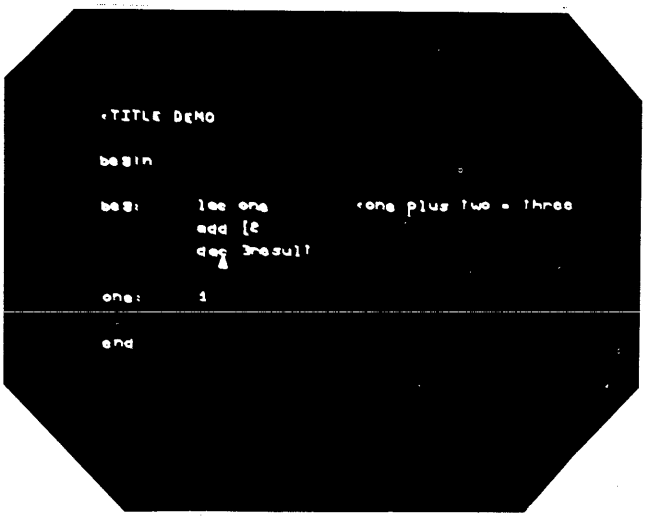


Figure 3—TVEDIT text for a demonstration program
This is assembly text for a simple program to add 1 and 2
The pointer under the 'c' indicates character mode editing

efficient for both the human user and the machine than alternatives involving escape characters or light pens; it allows the carriage return, space bar, and backspace to be used as control characters in a very natural way.

A pointer symbol displayed with the text always indicates the spot at which editing activity will be applied, and also indicates the line/character mode status of the editor. The user can set the pointer to an arbitrary page or move the pointer by lines or characters in any direction. Commands which would move the pointer off of the screen cause a new display "window" to be generated, keeping the pointer in view at all times. Windows usually contain 15 to 20 lines of text. Page divisions in the text are entirely under control of the user, and bear no particular relation to disk records, display windows, or paper sizes.

Following is a very brief description of the complete command set:

n	*space bar*	Move pointer right n characters.
n	*backspace*	Move pointer up n lines or left n characters.
n	*carriage return*	Move pointer down n lines.
n	*K*	Kill (delete) n characters or lines starting at the pointer.
n	*G*	Go to page n.
P		Insert page mark above current line (page marks can be deleted with K).
I		Enter insert mode.
W		Get next display window (allows rapid serial scanning).
F		Finish. Terminates edit run.

The detailed operation of each command is highly context-dependent, with regard to both the current state of the editor and the text being edited. Instead of employing a multiplicity of hard to remember commands, similar functions are lumped into a single command code, and distinctions are made on the basis of factors obvious to the user. For example, "backspace" and "kill" operate by lines or characters, according to the current mode, which is clearly represented in the display. There are no explicit commands for setting line or character modes since this shift is implicit in certain instructions such as spacing into a selected line or entering line mode by using the carriage return. An individual command is limited by the text on which it operates, for example, one cannot space past the end of a line or kill more than a page of text in a single command. The backspace and carriage return normally affect only the pointer and do not change text. After an insert command the backspace deletes preceding characters and carriage returns may be inserted as text. Any command using the control button causes the editor to revert to the normal mode in which typed text replaces existing characters. This type of special-case complexity was deliberately added to make the editor behave in a more natural manner, rather then conform to a set of rigid definitions. Such loosely defined operations are practical only because of the highly interactive nature of the display.

The editor is efficient in usage of machine time, being neither compute-bound nor I/O bound. The average editing run produces a very light load on the system. The random access feature and the file organization greatly reduces the amount of disk activity.

Two hardware factors are worthy of comment in relation to text editing. Efficiency would be enhanced by automatic tab stops in the display equipment; we have to generate a carefully counted series of spaces to achieve presentable tabs. A much more serious problem arises from the basic structure of the character set. Our six-bit characters with separate codes for case shifts generate tremendous problems in all phases of text handling. We cannot emphasize too strongly the importance of a seven-or eight-bit character coding to allow case shift status to be an integral part of each character.

Text files on the disk are organized as a page directory record followed by text records of identical format. This enables the editor to go directly to the

proper record for each new page. Local relative line addressing is easily handled since each record contains forward and backward links to its logical neighbors, and lines within a record are indexed for either forward or backward scanning. Text is packed in serial order in each record, and the link information is hidden behind the end-of-record mark. This fact, together with the unique escape character introducing file control codes, makes it simple for a serial text processor, such as the assembler, to read the file. The explicit "tab" and "carriage return" codes in our character set allows a high density on the file; however, a certain amount of space is normally left free in each record to allow for minor expansions without incurring the cost of linking in an overflow record. This overflow process is invoked automatically when necessary, and the user need not be concerned about such factors.

Random access text editing causes one distinct problem which must be provided for. Upon system or program failure we need to be able to recover as gracefully as possible in the face of lost or improper linkage information. Such a clean-up program is provided, along with serial file read/write services. A merge program is also available for merging selected pages from any number of files.

Assembler

The main characteristics of the assembler were largely determined by conditions of the hardware and software environment in which it operates. The emphasis in the system as a whole on rapid, simple, symbolic debugging dictated that the assembler be as fast as possible, and that symbolic programs be easy to read and modify.

Identifiers may be of any length, with the first six characters unique. Two methods of commenting are provided: one is a single character (<) which causes the rest of the line to be taken as a comment, and the other is the Algol variety, beginning with the word 'comment' and terminated by the next semicolon.

The input format is flexible, involving no fixed columns or fields. Statements are separated by end-of-line, or within the line by semicolons. Spaces, other non-printing characters, and blank lines are ignored. The large character set is used to keep the appearance of the program neat and pleasing, and to avoid confusion by assigning each operation in the assembler to a unique character. Thus, whenever a character appears it has a unique function regardless of context.

Block structure is provided in the assembler for two main reasons. The first is economy of symbol table space. Since each identifier takes at least three 18-bit words, and since the assembler must operate entirely within one 4096 word block, the space recovered by purging local symbols at the end of blocks is very important in assembling large programs. Secondly, block structure facilitates the inclusion of symbolic library routines and the combination of independent programs. The block structure works as in Algol, except that no declarations are required (an important point in machine language code, which tends to have a large number of labels). Prefixing the defining occurrence of a symbol with the character (↑) makes a symbol visible outside the block in which it is defined, allowing subroutines with multiple entry points to be enclosed in a single block.

The requirement for maximum assembly speed suggested a single-pass assembler, and the desire to save symbol storage space with block structure necessitated this approach. The assembler is organized internally as a simple two-stack translator; it evaluates expressions which may contain all of the ordinary arithmetic and logical operators and assembles the values into computer words. In addition there are the usual collection of symbol-defining and assembly control operations, and a general purpose recursive macro processor.

If a symbol is encountered before its definition it is called a forward reference. The symbol is entered in the symbol table, along with the address of the location from which it was referenced. Additional forward references to such a symbol are stored as a linked list in a general storage area shared with the symbol table. As soon as the symbol becomes defined, a "fixup" is issued to every location on the symbol's list of forward references. A "fixup" is a direction to the loader to change the contents of a specified location. References to non-local symbols must be treated as forward references until the end of the block, since no declarations of local symbols are required.

The main symbol table is stored and searched linearly. This facilitates implementation of the block structure, and since most references are to local symbols, searches of the table are usually short. At the end of the block, all symbols local to it are removed from the table, and stored on a disk file for use by the symbolic debugging program.

This assembler is noticeably faster than its predecessors on the same machine, all of which were two-pass processors. It executes between 600 and 900 instructions per word of code assembled, and requires about 75 seconds to assemble itself.

The most interesting conclusions reached in our experience with this processor are that block structure can be very useful in an assembly language, that single-pass assemblers are more efficient than their

multiple-pass counter-parts, and that flexible, readable format of the source program is a great saver of time and frustration.

Debugging

THOR's primary debugging aid is called RAID. It occupies the upper three-eighths of the user's core and is used to monitor the user's program. RAID's display shows the contents of sixteen memory locations in the user's program and the state of his accumulator, in-out register and program flags. The contents of each location are given in both octal and symbolic reconstruction of the assembly text. This presentation is far more informative than the conventional computer console. The program in Figure 3 was assembled and loaded into core. Figure 4, "A RAID Display of a Demonstration Program" shows the result of its execution.

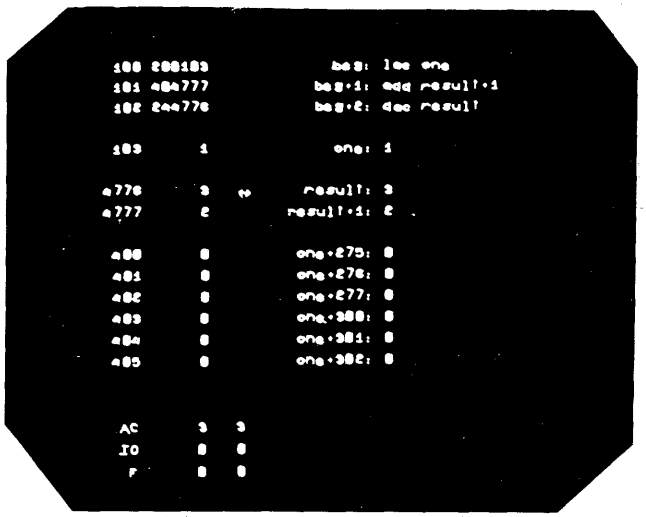


Figure 4—RAID display of a demonstration program
This is a display of the binary and symbolic of the simple demonstration program. The code was executed with the single step feature It left 3 in the accumulator (displayed near the bottom) and in the location 'result' (indicated by the pointer) The array of locations from 'one + 275' to 'one + 302' was displayed to illustrate the number of locations that may be simultaneously visible

RAID displays a pointer next to one of the sixteen locations to act as a focus of the user's attention. By typing single character commands the user can move the pointer to any location on the display. Other commands exist to modify the contents of any location or change the accumulator, in-out register and program flags. Still other commands enable the user to delete old locations and add new locations to the display. The command structure is designed to allow the user to change his focus of attention to the interesting parts of his program in a natural manner. He can trace the program flow, address chains, indirect references, and subroutine calls with a minimum of fuss.

The most useful features of RAID are the variations of single stepping. When the user single steps the instruction at the location indicated by the pointer the instruction is executed and the pointer moves to the next location in the program flow. All displayed locations and registers effected by the execution of that instruction are updated. The user may also plant breakpoints in his code. Normal usage is to plant a breakpoint just before a section of questionable code and start the program. When the breakpoint is reached, control passes to RAID, and all of the displayed locations are updated. The user then single steps through the questionable code, carefully observing the effects of each instruction. As soon as errors appear the user may investigate them immediately.

The single character control language allows skilled users to interact with RAID very rapidly. Typically bursts of such rapid activity will alternate with periods of thoughtful analysis. RAID's value as a self-instructional device is obvious: the novice programmer may enhance his understanding of the various computer instructions by executing them and observing the effects. The experienced programmer may occasionally revamp his understanding of a particular instruction. Generally a programmer single-stepping through his code will encounter occasions on which his image of the situation does not correspond to the actual situation. In writing code a programmer must anticipate the effects of the various instructions and he must maintain an image of what his code does to the memory registers involved. When debugging with RAID, he recreates this anticipated image, and can then correct his thinking where necessary.

The elimination of paper output and lengthy communication with the computer have made a great increase in the effectiveness and speed of debugging.

Usage

Flexible and efficient utility programs and basic system speed have given us a very fast edit—assemble or compile—debug cycle. This has introduced new programming habits. Programmers can now afford to edit and reassemble or compile to purge even moderately trivial bugs rather than make patches to octal code. The practice also helps alleviate the possibility of creating new bugs while correcting old bugs.

Our experience shows that it is easy to live without up-to-the minute listings of programs. One need only know the general position in the text of the proposed changes and a few TVEDIT commands will rapidly locate the desired area.

The fast edit—assemble—debug cycle pays other

dividends. We have observed a tendency toward composing programs on-line. The programmer describes an overview of his program, perhaps sketching out parts of the code, but he leaves the detailed coding for the console session.

THOR may be used to prepare batch processing jobs to be run on the IBM 7090. One method consists of writing the program using TVEDIT, and then converting the text to a disk file format compatible with the IBM processors. To use programs one submits a short job into the 7090 batch processing queue which calls the program from the disk.

Information may also be sent directly between the PDP-1 and the 7090 through a direct data channel. The PDP-1 interrupts the 7090 batch processor between jobs in a process known as time-stealing. Consequently, THOR users may prepare a TVEDIT image of the 7090 job, convert it to the BCD character code, and send it through the data channel. The 7090 sends output back which may be displayed immediately or placed in a disk file to be examined later.

A good example of a user program employing many of the features of THOR is our implementation of the Culler-Fried functional analysis display system. The left special control button was used to distinguish characters standing for operations from those standing for functions. Most operators and all functions are stored on the disk. Rapid disk access was critical in making the system practical.

The structure of the system was sketched out, but the majority of the code was composed at the console. About 4000 words of code were written and debugged in less than two man-weeks. A THOR display console was in use six to eight hours a day and the edit-assemble-debug cycle was constantly exercised. We feel that the Culler System would have taken at least twice as long to develop at a teletype console and months in a batch processing system. It has been impossible to gain exact statistics on the virtue of displays versus teletypes since no users could be coerced into using teletypes if display consoles were available. It seems that the teletype versions of most utility programs are harder to learn, less general, and more difficult to use.

The benefits of efficient and forgiving system and command languages cannot be overemphasized. If a time sharing system is to be used as a good debugging tool, the user must be able to spend long hours at a console without feeling frustration due to excessive waits or errors caused by either himself or the system.

The in-core system, system interpreter, and the error phantom can all be modified while the system is in operation. Thus partial system failures do not necessitate stopping the system. Naturally the normal THOR user may not be so omnipotent but by proper setting of the PDP-1 console test word switches any user program can attain privileged status. The system interpreter and error phantom were written and debugged as user programs with occasional sorties into privileged mode. Though a certain amount of care and caution must be exercised, this feature has proved invaluable.

Computer based teaching laboratory

One of the major projects under THOR has been a system of programs designed to teach mathematics and reading to elementary school children. The following is a brief outline of one of the programs, a drill program, and its use of THOR features:

The drill program gives practice in arithmetic and spelling skills. Twelve teletype consoles are located in local elementary schools. Each of the three hundred children in the experiment does twenty to thirty problems on the console in less than three minutes. Average response times are one to six seconds, so a fast system response time is necessary to keep up the pace. The consoles are placed in slave status to prevent the children from stopping the drills by calling the system interpreter.

The drill program is actually run by three user programs. One is a monitor which keeps a log of usage. The second is an elaborate report generator which may be called while a child is typing so that the teachers receive immediate data analysis of his progress and errors. The third program handles the typing of the problems on the teletypes, the receiving of answers, and data recording. The programs communicate with each other through the character switchboard, the extra user core memory, extra drum tracks available to user programs, and disk files modified while the programs are running. In short the drill programs make full use of the generality provided by THOR for program and console interaction.

REFERENCES

1 J McCARTHY et al
A time-sharing debugging system for a small computer
Spring Joint Computer Conference pp 51-57 1963

2 A KOTOK
DEC debugging tape
Memo MIT-1 rev MIT Cambridge Mass December 11 1961

3 J SAUTER
The Stanford University PDP-1 manual
Stanford Time-Sharing Memo No 36 August 30 1965

4 B TOLLIVER
TVEDIT
Stanford Time-Sharing Memo No 32 March 1 1965

5 P STYGAR
RAID
Stanford Time-Sharing Memo No 37 Nov 2 1965

6 B W LAMPSON
Interactive machine language programming
Fall Joint Computer Conference pp 473-481 1965

7 B D FRIED
The STL On-Line Computer
Vols 1 and 2

8 G J CULLER
Function Oriented On-Line Analysis
Workshop on Computer Organization pp 191-213 1962

9 J GILMORE
Lincoln Lab memo out of print

10 G STRACHEY
Time-sharing in large fast computers
in Proceedings of the International Conference on Information Processing pp 336-341 UNESCO Paris 15-20 June 1959 UNESCO Paris 1960

COMPOSE / PRODUCE: a user-oriented report generator capability within the SDC time-shared data management system

by WILLIAM D. WILLIAMS*
Shell Oil Company
New York, New York

and

PHILIP R. BARTRAM
System Development Corporation
Santa Monica, California

INTRODUCTION

At present there are numerous report program generators, most of which are quite similar in purpose and in method of operation. Although they generally relieve the programmer from concern with detailed program logic, they nevertheless require him to have intimate knowledge of the characteristics of his data base, the kinds of entries he wants extracted, the manipulations to be performed, and the detailed format of the desired report. Furthermore, as report program generators have become more generalized (and thus more flexible), they have also become more complex, requiring the user to furnish complicated specification statements in coded form. This, of course, demands an increased amount of training and experience on the part of the programmer, and makes his source programs more subject to error.

In an attempt to overcome some of these difficulties, System Development Corporation is developing a user-oriented report program generator as part of the Time-Shared Data Management System (TDMS). As shown in Figure 1, the report generator function within TDMS is a two-phase program: the first phase, called "COMPOSE," is used to design the report on-line; the second phase, called "PRODUCE," actually produces the report.

TDMS is a general-purpose file management system operating under the SDC time-sharing executive on IBM S/360-series computers. Development of the system is being sponsored partly by the Advanced Research Projects Agency of the Department of Defense. As with any data management system, the

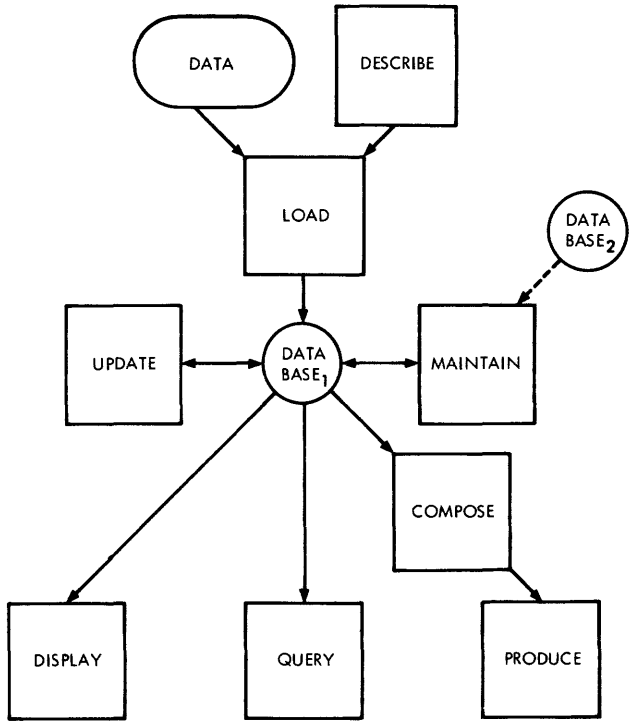


Figure 1 – TDMS system components

*Rearch Associate at System Development Corporation

functions to be provided by TDMS include the ability to describe and load a data base, maintain the data base, and retrieve data in response to human query.

This paper discusses the design features of on-line report program generators, and gives several examples of the operation of COMPOSE and PRODUCE. A more detailed discussion of the logical structure of a TDMS data base was given by Emory W. Franks at the Spring Joint Computer Conference, Boston, Massachusetts, April 26, 1966.*

Report generation

For purposes of discussion, "report" is defined as a display of information, the volume of which is such that most of the time the end product will be produced on an off-line printer. There are essentially two steps that are necessary to produce a final report: report description and report generation. In an on-line, time-shared environment, these two steps are so distinct that it is convenient to treat them as separate entities. When a user is describing his report, there is a great deal of on-line dialogue that must take place between him and the system; but once the description is complete, the actual generation of the report is a production-like job that requires little or no user interaction.

Thus, in the design of the TDMS report generator, these two functions were kept separate. The flow of information for COMPOSE and PRODUCE is shown in Figures 2 and 3. Functions performed by COMPOSE and PRODUCE are listed below:

COMPOSE

- Obtains a data base name on-line from the user.
- Creates a report description, interactively, from on-line statements by the user.
- Saves the report description under a file name given by the user; this will be its means of later communication with PRODUCE.

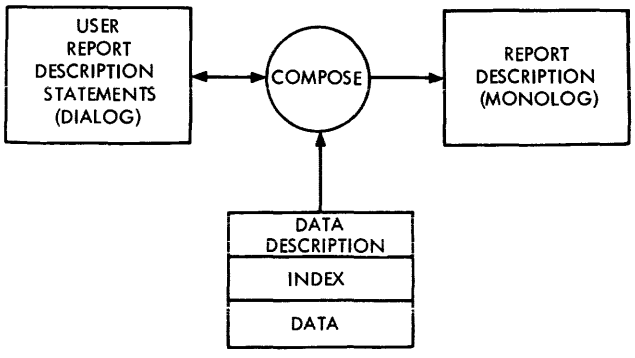


Figure 2 – Information flow diagram for COMPOSE

*See E. W. FRANKS, *"A data management system for time-shared file processing using a cross-index file and self-defining entries,"* AFIPS Conference Proceedings, Vol. 28, 79-86, 1966.

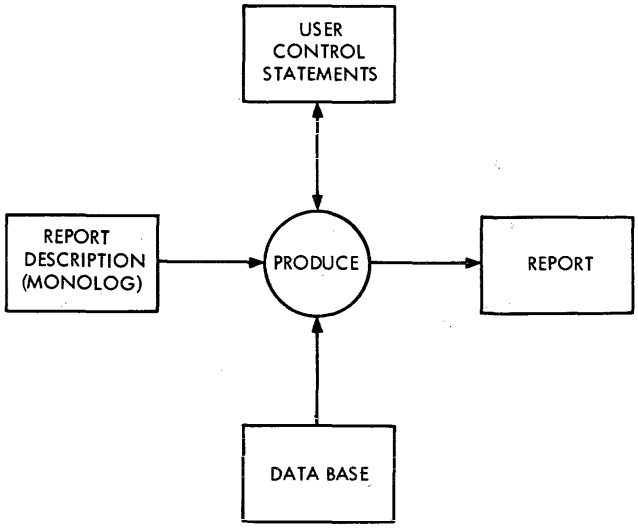


Figure 3 – Information flow diagram for PRODUCE

PRODUCE

- Obtains a report file name on-line from the user.
- Queries the user concerning any incomplete information from the COMPOSE phase.
- Obtains the report description and the related data base from the system.
- Compiles a report program.
- Generates a report by operating the compiled program.

Design criteria for a report generator

The main limiting factors in the design of report generators in the past have been the machine configuration the designers had to work with, and the logical structure of their data files. The report generator discussed in this paper is designed to operate under the control of a time-sharing executive using IBM S/360 computers (Model 50H or larger). File organization in TDMS makes optimum use of random access mass memory, permitting rapid retrieval and the maintenance of hierarchical relationships among file entries. Thus the designers of COMPOSE and PRODUCE were able to take advantage of facilities that were not available to many previous designers of report generators.

The main design goal of most report generators is to provide the capability for rapid, economical retrieval and presentation of any data from a data base. However, many report generating programs – although economical of machine time – largely ignore the specific needs of the user. Since the ultimate user of the reported information is concerned mainly with receiving the right amount of information, at the right time, and in a *usable* form, it seems natural to assume that the more directly involved the user becomes in the whole report generation process, the greater the

probability that his needs will be met. Furthermore, in order to accommodate the wide variety of users who have access to a time-shared system, it is necessary to provide a means of communication with the system that is simple and natural for the majority of users.

These considerations led the designers of the COMPOSE/PRODUCE report generator to design a system that involves the user—at least in the COMPOSE phase—to a greater extent than has been thought possible before. The following subsections describe how this plan was implemented.

User-oriented system

Apparently, the most desirable form of communication between a user and his data base is natural English; research is being done by several organizations in that regard. For example, an SDC-developed forerunner of TDMS, called TSS-LUCID, uses a command language which is a restricted subset of natural English. TSS-LUCID met with such favorable user response that the language of COMPOSE has been designed along the same lines. Within TDMS, the language specifications are such that they are consistent over all operations of the system from data description to retrieval.

User concern with the logical procedure of generating a report is kept to a minimum in COMPOSE. The user describes his report in a random order of nonprocedural statements. COMPOSE interprets these statements and builds the necessary report description for the PRODUCE phase of the operation.

After routine initial dialogue (naming files, etc.), the user of COMPOSE employs four basic types of descriptive statements corresponding to the "command" words shown in Table I. Group I statements determine selection, order of retrieval, and data reduction to be done. Examples of Group I statements are shown in Figure 4. Group II statements describe the contents of the report itself; examples of Group II statements are shown in Figure 5. Group III and IV statements give the user the opportunity to receive (on-line) a picture of how his report will be formatted. He may call for one of two types of review at any time. He also has the ability to add, delete, change or reformat his report, as is shown in Figure 6.

GROUP	I	II	III	IV
STATEMENT TYPE	DATA BASE SELECTION	REPORT DESCRIPTION	REPORT REVIEW	FORMAT CONTROL
STATEMENTS	QUALIFY SORT DERIVE	TITLE HEADING CONTENT RECAP	REVIEW PROOF	MASK FEED SPACE LIMIT PUT REPRINT

Table 1—Basic compose descriptive statements

```
QUALIFY GAME WHERE FIELD EQ HOME

SORT BY TEAM, PLAYER

DERIVE PCT = (HIT/AB) * 100
```

Figure 4—Examples of data base selection

```
TITLE IS SEASON PERFORMANCE 1966

HEADING IS CITY, TEAM, SEASON HISTORY

CONTENT IS GAME, OPPONENT, FIELD, GATE

RECAP ON PLAYER = SUM HITS, SUM RUNS, MAX RBI

RECAP ON TEAM   = 'L TOTALS, SUM TMRUNS, SUM OPRUNS
```

Figure 5—Examples of Report Description

```
REVIEW REPORT

PROOF T1 . . . T4

FEED PAGE AFTER RECAP ON PLAYER

LIMIT OPPONENT = H9

PUT T1 . . . T4 AFTER OVERFLOW
```

Figure 6—Examples of Report Review and Format Control

Flexible use of TDMS data bases

Without describing in detail the logical structure of a TDMS data base, suffice it to say that initially the user describes what he considers to be a logical entry. This entry may consist of any number of names,

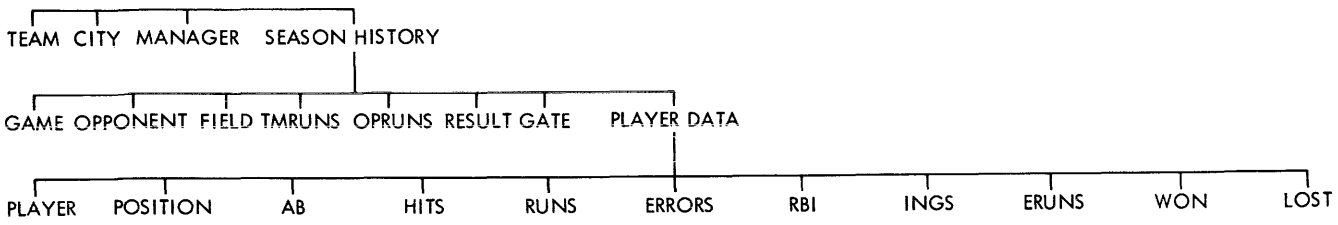


Figure 7—Logical structure of a typical TDMS data base entry

dates, and numbers. Values which occur only once per entry are said to be at level 0; all others are defined as members of branches of a logical tree structure. These branches are called "repeating groups" and allow up to 16 levels of subdivision. Figure 7 shows a simple illustration of an entry for a baseball team season history.

If a user were to request a report from this data base, he might make a statement as follows:

CONTENT IS TEAM, COUNT WON, COUNT LOST

At report generation time, PRODUCE will loop through each *entry* and print the won-lost record of each team. PRODUCE is said to be iterating on a cluster of data values, the initial definition of a cluster being a logical entry. COMPOSE allows the user to be much more flexible than this. By the use of an operator known as "Qualify," he may define his own cluster:

QUALIFY OPPONENT

This now causes the iteration to take place on each occurrence of "Opponent" and all related values both up and down the logical tree. "Qualify" not only allows the user to define his own cluster, but gives him the opportunity to subset his data base:

QUALIFY GAME WHERE FIELD EQ HOME

The iteration will take place on each occurrence of "Game," but only on those games played at home.

Once a cluster has been defined, the data base, or its qualified subset, may be sorted on any values in that cluster and in any order. Although "Game" was originally defined as being minor in logical relation to "Team," TDMS allows the user to make "Team" or "Opponent" (or other element names on those respective levels) minor in relation to "Game," e.g.:

SORT BY GATE, OPPONENT, TEAM

Interaction with the user

The ability of the system to interact on-line with the user is of enormous value. COMPOSE attempts to achieve the following in its dialogue with the user:

- Inform the user of absolute error conditions.
- Warn him of potential errors.
- Prompt the user about rules concerning his data base or a particular facet of COMPOSE.
- Alert him concerning incomplete information.

Data reduction capability

COMPOSE gives the user ability to do data reduction in two ways. The first involves an element name from the data base, modified by one of the following: Sum, Average, Count, Maximum, Minimum, Sigma. The range over which these operators have effect is under complete control of the user.

The second form of data reduction involves the derived variable, which may be any arithmetic expression, including one or more related element names from the data base or additional derived variables. Figure 8 shows a simple example of data reduction and the use of dialogue between the system and the user. When COMPOSE encounters an element tag that does not appear in the data base, it requests a definition. If the definition is a valid form of data reduction, COMPOSE will request a format for the derived variable. (In the following figures, the system's response is underscored; the user's input is not.)

CONTENT IS TEAM, WINS, LOSSES, PERCENTAGE

DERIVE WINS =: COUNT RESULT

DERIVE LOSSES =: (162-WINS)

MASK LOSSES =: 099

DERIVE PERCENTAGE =: (WINS/162)

MASK PERCENTAGE =: .999

Figure 8—Examples of system-user interaction

User control over format

COMPOSE has access to certain tables in a user's data base that contain information regarding size, format, etc., of all the data fields. Using this information, and setting up arbitrary standards such as centering of title lines, alignment of heading and content lines, etc., COMPOSE attempts to format the report as described by the user, although the user does not

have to worry about field or line position or about the order in which he entered his descriptive command statements. He may get an excellent idea of what his report will look like by use of one of the report review commands. They are:

- REVIEW, in which he merely receives an ordered monologue of what he has said.
- PROOF, in which a sample output of the report is shown.

The user may call for one of these reviews at any time during COMPOSE. If he wishes to alter the format of one or more fields or lines, he may do so by using SPACE (horizontal) or FEED (vertical).

Figure 9 shows what the user would receive as an on-line response for the given content statement, and an edit statement he might make.

Changing, updating and completing a report

COMPOSE allows the user to easily change any one of his statements either during his initial use of COMPOSE or at a later date. He may also call on a previously described report and use it as a basis for composing a new report without destroying his original. COMPOSE has also given the user the ability to leave certain decisions unresolved, to be completed later at the time of actual report generation (PRODUCE). This has been done by use of 'K preceding a word. For example:

```
CONTENT IS TEAM, OPPONENT, GATE
PROOF
1.....10.....20.....30.....40.....50.....60...65
      AAAAAAAA    AAAAAAAA    99999
SPACE 10 AFTER OPPONENT IN CONTENT IS TEAM
```

Figure 9—On-line content statement response

```
QUALIFY PLAYER WHERE SALARY
    GR 'K SAL
```

By leaving the "Salary" limit open until the time of PRODUCE, this one report description can be used for any number of reports with different "Salary" criteria.

Sample uses of COMPOSE

Using the data base described in Figure 7, two simple problems, report descriptions and sample output are shown.

Problem 1

Generate a report showing the record of home attendance for each team, ordered by opponent. The report is to show the home team, game number, and attendance. Show the average attendance for the season, broken down by opponent; also show the average attendance for the season for all opponents. The record for each home team is to begin on a new page. Figure 10 shows the user statements, and Figure 11 shows a sample output.

```
TITLE IS HOME ATTENDANCE 1966
QUALIFY OPPONENT WHERE FIELD EQ HOME
HEADING IS HOME TEAM, OPPONENT, GAME NO., ATTENDANCE
SORT BY CITY, OPPONENT
CONTENT IS CITY, OPPONENT, GAME, GATE
RECAP ON OPPONENT = 'L AVERAGE ATTENDANCE AGAINST, OPPONENT, AVG GATE
RECAP ON CITY = 'L AVERAGE ATTENDANCE FOR SEASON, AVG  GATE
FEED PAGE AFTER RECAP ON CITY
```

Figure 10—User report description for problem 1

Problem 2

Generate a report showing the performance of each pitcher, broken down by the opponents he has faced, and for each opponent give the number of innings he pitched, his won-lost record and earned-run average. Show each pitcher's season record for the same. Begin each player's report on a new page. Figure 12 shows the user statements, and Figure 13 shows a sample output.

SUMMARY AND CONCLUSION

In order to provide a report generation capability within a data management system that operates under time-sharing, it is necessary to design a report program generator that provides sufficient power to meet the user's needs, but that is not so complex that the nonprogrammer user will be deterred. The COMPOSE/PRODUCE program being developed at SDC is a step toward the achievement of this goal. It allows the user to describe his report on-line, to do data reduction, and to control output format. It takes advantage of the advanced file structure of a TDMS data base, permitting great flexibility in the selection and representation of various reports from a data base. The COMPOSE portion of the program accepts command statements in an English-like language and in any sequence, in order not to deter the user who is more familiar with his data base than he is with the mechanics of file manipulation and report generation. Work is continuing at SDC to make user communication with large, structured files of data as simple and logical as possible, while retaining the power made available to the user through large-scale computing equipment.

HOME ATTENDANCE 1966

HOME TEAM	OPPONENT	GAME NO.	ATTENDANCE
LOS ANGELES	ATLANTA	4	23468
		5	20182
		6	26437
		51	31628
		52	28727
		98	19605
		99	17342
		143	24471
		144	25389
		AVERAGE ATTENDANCE AGAINST ATLANTA	24138
LOS ANGELES	CHICAGO	11	16237
		12	15331

Figure 11 – A portion of final report for problem 1

```
TITLE IS PITCHING PERFORMANCE 1966
QUALIFY PLAYER WHERE POSITION EQ PITCHER
SORT BY TEAM, PLAYER, OPPONENT
HEADING IS TEAM, PITCHER, OPPONENT, INNINGS, WON, LOST, ERA
RECAP ON OPPONENT = CITY, PLAYER, OPPONENT, SUM INGS, SUM WON, SUM LOST, ERA
DERIVE ERA =: 9* ((SUM ERUNS)/(SUM INGS))
MASK ERA =: 09.99
RECAP ON PLAYER = SUM WON, SUM LOST, AVG ERA
FEED PAGE AFTER RECAP ON PLAYER
```

Figure 12 – A portion of final report for problem 2

PITCHING PERFORMANCE 1966

TEAM	PITCHER	OPPONENT	INNINGS	WON	LOST	ERA
LOS ANGELES	DRYSDALE	ATLANTA	96	3	3	3.65
		CHICAGO	84	2	4	3.26
		CINCINNATI	103	2	1	2.64
		HOUSTON	114	3	1	1.78
		NEW YORK	124	2	0	2.46
		PHILADELPHIA	83	1	4	3.43
		PITTSBURGH	78	2	2	2.85
		SAN FRANCISCO	97	0	4	3.96
		ST. LOUIS	108	1	2	2.77
				16	21	3.13

Figure 13 – A portion of final report for problem 2

Code generation for PIE (parallel instruction execution) computers

by J. F. THORLIN
CONTROL DATA Corporation
Palo Alto, California

INTRODUCTION

Computers capable of executing more than one instruction at a time, e.g. buffered I/O instructions executing in parallel with computation, have been available for many years. Only recently have machines been generally available which possess the PIE characteristic to any higher degree, most notable of these is the CONTROL DATA® 6600 Computer. Due to the lack of such machines in the past, little work has been done to develop code generation strategies which are capable of making effective use of this facility.[1,2,3,4] The techniques used by a hand coder to generate good code for a PIE computer differ significantly from those used on serial instruction computers; they are much more time consuming and require more knowledge of the hardware, but the results will more fully utilize the available circuitry and will, consequently, produce significant improvements in computing speeds. The reordering of old code produced with earlier code generation methods has reduced execution time by 50 per cent.

Environment

This paper deals with the production of good code for a PIE computer, the CDC 6600, from an artificial algebraic language, FORTRAN. The CDC 6600 central processor has eight 60-bit X registers for arithmetic, eight 18-bit B registers for counting and indexing, and eight 18-bit A registers for referencing memory. A1 through A5 are used for loading the associated X register; similarly A6 and A7 are used for storing. B0 contains a constant zero. A0 can be used like a B register insofar as it does not affect memory. The arithmetic portion of the computer has ten function units which may be operating simultaneously. There are two load/store (increment) function units (I1,I2), two floating multiply units (M1,M2) and one unit each for floating divide (DV), integer add (LA), floating add (FA), Boolean (BL), branch (BR) and shift (SH) functions.

Method

The method is best described with an example. A group of FORTRAN source statements is shown in

```
      IF   (ATAG   .LT. B*B)   CALL COMET
     [PSI = (B+ACT   (N)   **2 + RHO * SIGMA  (N)
     |K = N+K
     |QTAB  (N) = XTAB (I)/RHO +PSI
4    TAB2  (2, 2*K) = PSI + ATAN (RHO)
```

Figure 1 – Source statements

Statements within the bracket constitute a flow block or sequence. Initially, these statements are analyzed and converted to a register free notation call R-list which would appear as is shown in Figure 2.

$R_1 \leftarrow$ B	$R_{11} = R_6 + R_{10}$	$R_{19} = R_{17} / R_{18}$
$R_2 \leftarrow$ N	$R_{12} =$ N (R_{11})	$R_{20} \leftarrow$ PSI
$R_3 \leftarrow$ ACT - 1, R_2	$R_{12} \rightarrow$ PSI	$R_{21} = R_{20} + R_{14}$
$R_4 = R_1 + R_2$	$R_{13} \leftarrow$ N	$R_{22} =$ N(R_{21})
$R_5 =$ N (R_4)	$R_{14} \leftarrow$ K	$R_{23} \leftarrow$ N
$R_6 = R_5 * R_5$	$R_{15} = R_{13} + R_{14}$	$R_{24} \rightarrow$ QTAB-1, R_{23}
$R_7 \leftarrow$ N	$R_{15} \rightarrow$ K	
$R_8 \leftarrow$ SIGMA-1, R_7	$R_{16} \leftarrow$ I	
$R_9 \leftarrow$ RHO	$R_{17} \leftarrow$ XTAB-1, R_{16}	
$R_{10} \leftarrow R_8 * R_9$	$R_{18} \leftarrow$ RHO	

Figure 2 – Generated R-list (N(R) indicates the normalization of the result of a floating add or subtract; left arrows are loads and right arrows are stores)

The generated R-list is then scanned and common suboperations are eliminated resulting in the squeezed R-list shown in Figure 3.

From the squeezed list a PERT-like network, the dependency tree, is formed showing the precedence of

1.	$R_1 \leftarrow B$	(8)	11.	$R_{12} = N(R_{11})$	(4)
2.	$R_2 \leftarrow N$	(8)	12.	$R_{12} \rightarrow PSI$	(10)
3.	$R_3 \leftarrow ACT\text{-}1, R_2$	(8)	13.	$R_{14} \leftarrow K$	(8)
4.	$R_4 = R_1 + R_3$	(4)	14.	$R_{15} = R_2 + R_{14}$	(3)
5.	$R_5 = N(R_4)$	(4)	15.	$R_{15} \rightarrow K$	(10)
6.	$R_6 = R_5 * R_5$	(10)	16.	$R_{16} \leftarrow I$	(8)
7.	$R_8 \leftarrow SIGMA\text{-}1, R_2$	(8)	17.	$R_{17} \leftarrow XTAB\text{-}1, R_{16}$	(8)
8.	$R_9 \leftarrow RHO$	(8)	18.	$R_{19} = R_{17} / R_9$	(29)
9.	$R_{10} = R_9 * R_8$	(10)	19.	$R_{21} = R_{12} + R_{19}$	(4)
10.	$R_{11} = R_6 + R_{10}$	(4)	20.	$R_{22} = N(R_{21})$	(4)
			21.	$R_{22} \rightarrow QTAB\text{-}1, R_2$	(10)

Figure 3 – Squeezed R-list

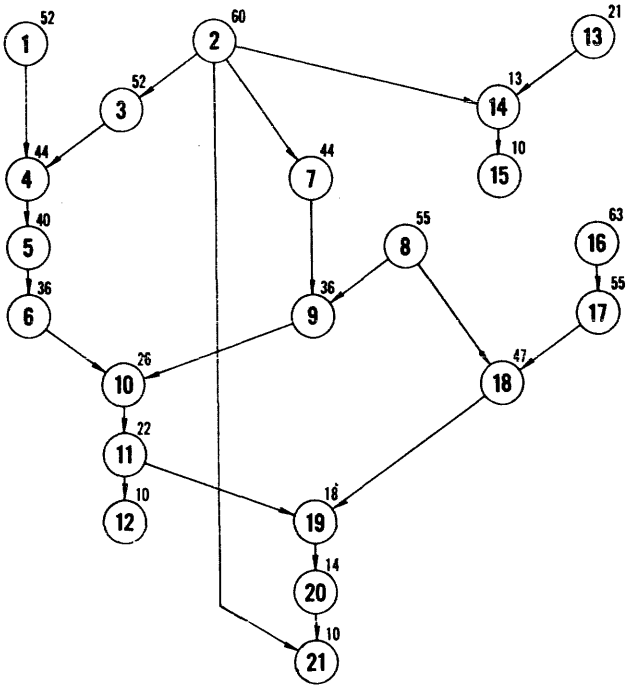


Figure 4 – Dependency tree

operations.

The numbers within the circles at the nodes are keyed to the R-list ordinal in Figure 3. The time in machine cycles required for each operation is known and is shown enclosed in parentheses following the R-list entry in Figure 3. From this information, the latest time at which each operation must begin in order to finish executing the network in the minimum amount of time is calculated. This is done assuming no conflicts of any kind and parallel instruction issue as well as execution. These times, called priorities, are the numbers shown next to each circle; in a PERT sense they correspond to negative late start times with the network being completed at time zero. Code is generated beginning with the highest instructions it is required that the preceding operations have been issued and there are no function unit or register conflicts; for this purpose, a picture of the status of all registers and function units must be maintained. Using this approach, the code shown in Figure 5 is produced resulting in the indicated overlap of operations.

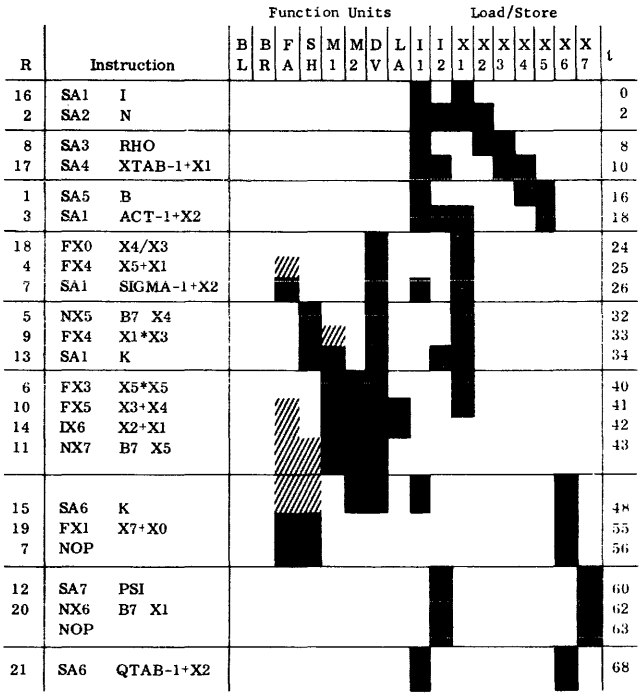


Figure 5 – Generated code and timing (it is the time at which the associated instruction is sent to its function unit. On time unit, minor cycles, equals one hundred nanoseconds)

CONCLUSION

Had this machine been a strictly serial processor we can see, by adding the numbers in parentheses in Figure 3, that the code would have taken 170 cycles to execute. By having the parallelism and taking advantage of it we can go on to the next operation as early as time 70.

The example makes apparent the power which is available in a PIE computer; the method described takes advantage of this power.

REFERENCES

1 HARRY L. NELSON
Program Optimizing Techniques for the CDC 6600 Central Processor
Technical Report April 6 1965

2 K A WOLF R W ALLARD and R A ZEMLIN
Some Effects of the 6600 Computer on Language Structure Communications of the ACM VOL 7 No 2 February 1964 112-119

3 *SAM F MENDICINO and RICHARD G ZWAKENBERG A FORTRAN Code Optimizer for CDC 6600*
Technical Report April 1965

4 J SCHWARTZ
Large Parallel Computers
Journal of the Association for Computing Machinery Vol 13 No 1 January 1966 25-32

5 Control Data 64/6600 Computer Systems Reference Manual Pub No 60100000 Rev B September 1966

SNUPER COMPUTER—a computer in instrumentation automaton*

by G. ESTRIN, D. HOPKINS, B. COGGAN and S. D. CROCKER
Department of Engineering
University of California Los Angeles
Los Angeles, California

INTRODUCTION

Classically the preponderance of measurements made on information processing systems is done for purposes of prediction or diagnosis of malfunction. The environment is monitored with observations of the ambient temperature, common system voltages and currents, and clocking pulse streams. Micro-measurements of system elements are made on-line with oscilloscope or meter and off-line by specialized more complex testing systems. Systems are used to test themselves by generation of diagnostic programs using predefined data sets and by explicit controls permitting degradation of the environment. As opposed to equipment malfunction, programs malfunction only in the sense that there is an inconsistency between the intent of the programmer and the finally executed machine code. Diagnostic programs aid in making measurements related to consistency.

Some measurements of normal system behavior are also in use. Built-in accounting and analysis of system logs are used to provide a history of system performance as well as establish a basis for charging users. Programs are executed in an interpretive mode to record less accessible behavior. Relative system performance is measured using standardized problem mixes applied to a working system or a simulated system. More sophistricated approaches to system instrumentation are found in the IBM hardware monitor system,[1] memory utilization studies by Graham at Rice,[2] and work carried out in time sharing environments at SDC[3-7] and M.I.T.[8-11]

This paper focusses attention on the broad question which may be stated in the form, "By what means and to what extent is it possible to measure behavior of users as reflected through terminals, user programs, system programs and equipment in information processing systems which may in general reach the complexity of remote, multiuser, multiprogrammed, multilevel, distributed, general purpose multiprocessor systems?"

During the past two decades advances in computer technology have occurred in such large steps that the marketplace has accepted microscopic measures such as component propagation delay or memory cycle time on the one hand, or gross measures such as the total system cost or individual problem solution time on the other. An engineer who wishes to concern himself with performance criteria in the synthesis of new systems is frustrated by the weakness of measurement of computer system behavior.[15] A user who wishes to generate efficient programs to implement an algorithm stated within his problem context finds a dearth of tools.

The following paper establishes a foundation for continued study and experimentation. Although there are a number of deep questions to be considered concerning the goals of instrumentation in the hands of the problem solver, this paper concentrates primarily on objective measurements of system performance. Part I characterizes information processing systems, discusses classes of measurements, and describes artifact introduced by instrumentation. Part II presents system approaches to the instrumentation problem and describes the characteristics of a system called SNUPER COMPUTER under development at UCLA.

*This work was supported by the Advanced Research Projects Agency, the Atomic Energy Commission, Division of Research [AT(11-1) Gen 10] and the Office of Naval Research, Information Systems Branch [Nonr 233(52)].

PART I

Information processing systems consist of equipment providing a basic instruction set and resident programs providing an extended instruction set. The basic instruction set wired into an information processing system permits automatic arithmetic and logical transformations on operands, transmission to and from the external environment, and automatic alteration of a normal sequence of events conditional upon the results of transformations or upon the state of the system. The basic instruction set is greatly expanded by maintaining frequently used program sequences resident in storage. This evolution first took the form of development of libraries of subroutines which could automatically be called to execute transformations. In today's complex systems, resident routines play the functions of gathering multiple input streams, distributing multiple output streams, translating from higher languages to machine code, allocating storage to programs, allocating tasks to independent parts of the system, recording time of a problem in the system and carrying out error detection and correction.

We now consider models of systems and models of the computations flowing through them. In each case we seek to delineate measurement questions.

Simple sequential batch processing system

In a simple sequential batch processing system, problem analysis and choice of algorithm are followed by "off-line" program preparation in system compatible form. A queue of user programs is formed off-line and the queue may be rearranged according to priorities. Each one of the user's programs takes possession of the system until completion of the task. A program and data stream are entered into the system. The program is initiated and stops when it has completed generation of an output file whereupon the next user program takes ownership. Simple models of the process and the system are illustrated in Figures 1 and 2.

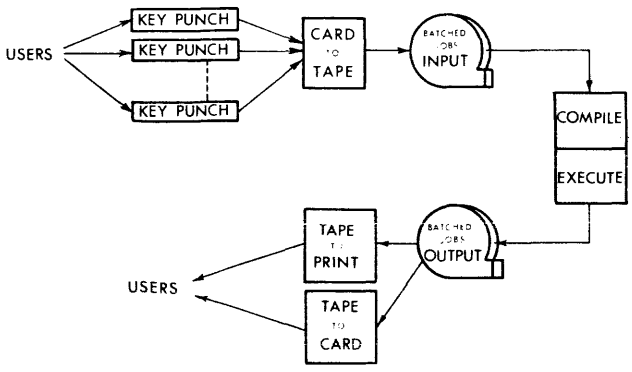


Figure 1 – Sequence of activities in a batch processing system

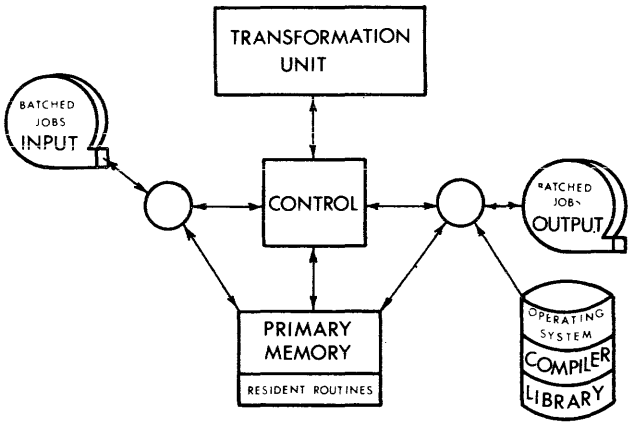


Figure 2 – Model of a batch processing system

We now turn our attention to types of system behavior which we may be interested in measuring.

The stream

If we look on the system as acting on a stream of independent programs we can seek measures of performance for the separate system components. What percentage of the time is each component active? What percentage of storage capacity is utilized? What is the frequency distribution of instruction types? What is the frequency distribution of the extended instruction set resident in the library or in the operating system as a whole? What percentage of time is spent in operating system manipulation as opposed to execution? The stream may consist of a set of programs representative of a computing center's activity or an artificial mixture of instructions and operands.

The particle

If we look on a segment of computation as a particle in the stream we can seek to track the particle and measure properties of the computational sequence as well as all of the system properties above. What are the branching statistics at decision points in the segment? How many iterations occur in data dependent loops? What are the most time consuming parts of the computation? Is the actual sequence of steps consistent with the program which was written?

Characteristic of all measurement processes is an artifact introduced by the measurement itself. Let us consider the range of measurement systems, procedures and associated artifact.

Self measurement

Given a universal computer it is possible to simulate any other computer. Furthermore it is possible to execute a stream of computations or track a particular sequence in an interpretive mode on the simulated machine. Thus an operation code in any instruction may be transformed into an address calling

out a routine which executes the corresponding predefined operation on predefined operands and records measurements of properties of a model of the system under study. The model may be a crude one or it may seek to represent details as fine as the rise time of state changes under varying information conditions. The artifact introduced by this process is an extreme change in the time required to do the computations being subjected to measurement. The cost of the measurements and the difficulties involved in writing simulation programs have made extensive use of such methods rare. They have been applied in numerical experiments to guide costly system decisions such as increase in the complexity of anticipatory control,[12] effectiveness of parallel processing,[13] and effectiveness of multiprogramming storage allocation and job scheduling strategies.[3,14,15]

The artifact may be reduced in three ways:

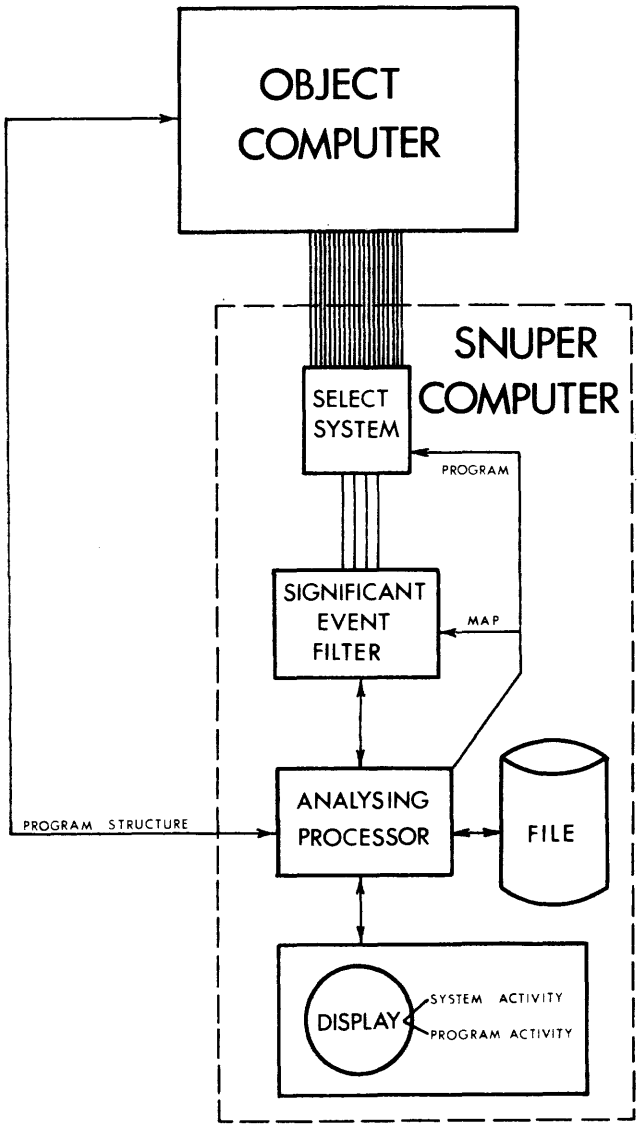


Figure 3 – Model of an instrumentation automaton

1. Instead of observing every operation, make measurements on gross events.
2. Selective injection of measurement flags into programs by the operating system such that artifact is introduced only at measurement points in programs rather than with every operation.
3. Attachment of external measuring systems which can observe on-going processes.

External measurements

Running time meters on system components are examples of special purpose instruments. The IBM Hardware Monitor[1] is an example of a general data acquisition system seeking to capture a continuous stream of events for off-line analysis.

Figure 3 illustrates the system properties of a general purpose instrumentation automaton. There is an extremely large set of points which may be observed in an object computer. We assume that a subset of these is brought out for possible observation. The Selection System must be capable of executing a program to select desired observation points at times when their states are of interest. The outputs may be delivered directly for CRT display on a recurrent time axis. For most of the measurements considered here they proceed through the next stage of the system which must filter the flow of events observed in the object computer so as to reject "uninteresting" events and present to an analyzing system a data rate which it can reasonably process. We call this a "Significant Event Filter" which might hold a program generated map of measurements and reject all irrelevant occurrences. A direct path between the object computer and an analyzing processor is required to deliver information about the structure of the program to be instrumented. The analyzing processor in turn can bring results of analysis and control to the object computer. A file for storage of significant data and a display system for presentation of reduced data in a form absorbable by humans complete the requirements.

Figures 4 and 5 present hypothetical display formats which would provide responses to measurement questions raised above.

Figure 6 illustrates a model of a multiprocessor system and Figure 7 gives examples of desired measurements.

Figure 8 illustrates a model of a remote multiuser, multiprogrammed system and Figure 9 pictorially illustrates additional desired measurements.

We now consider one display in more detail to bring out properties of the graphics station. Figure 10 shows an example of an instruction frequency display. The named functions along the bottom line

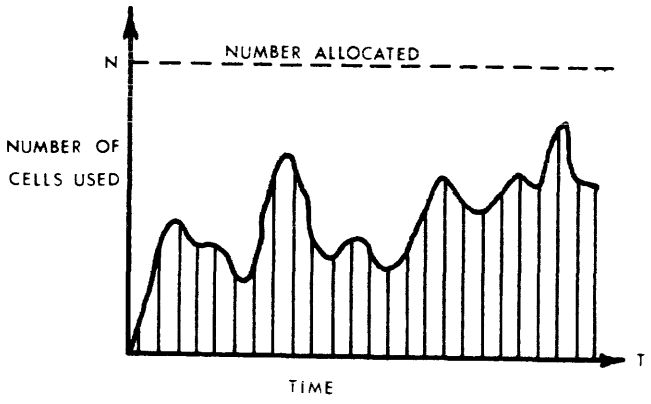
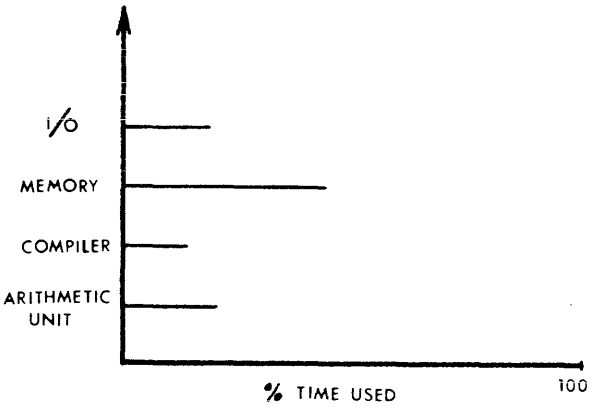
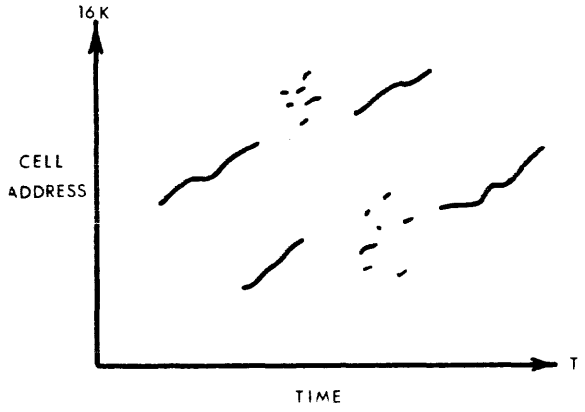


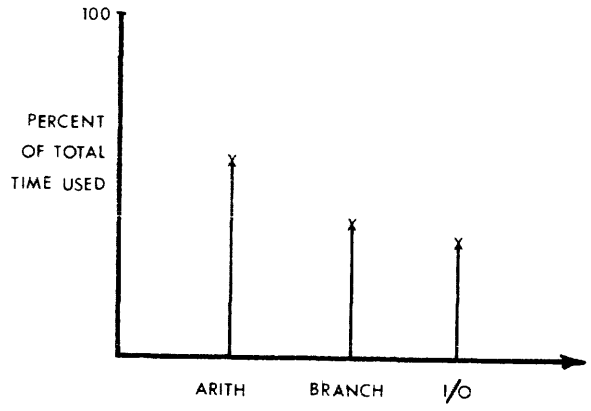
Figure 4—**Conceptual displays of system utilization**
(A) Memory profile



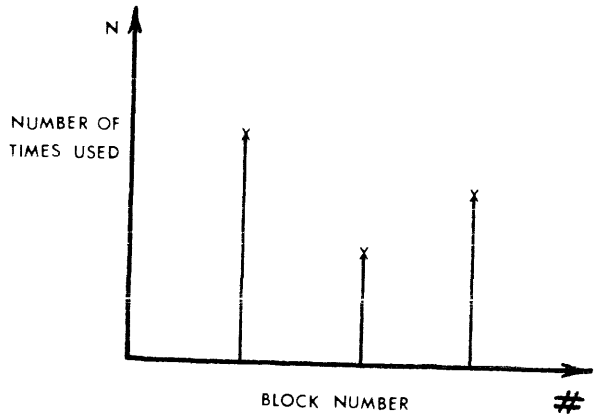
(D)Percent utilization



(B)Memory activity



(E)Instruction class statistics (ordered)



(C)Memory block usage

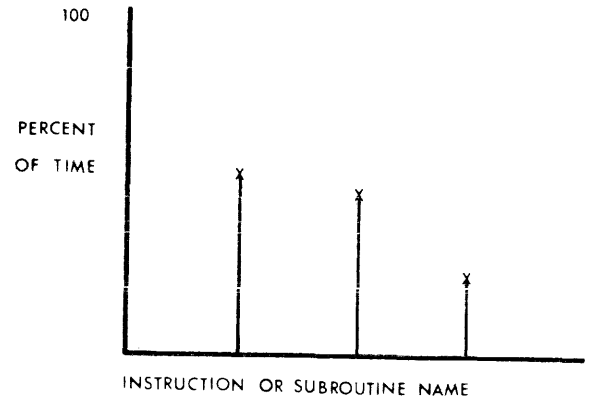
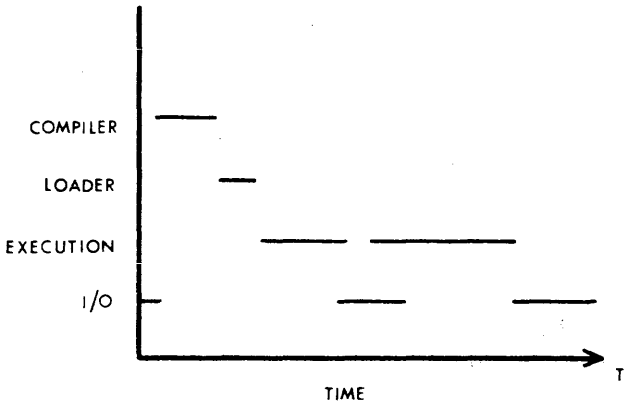
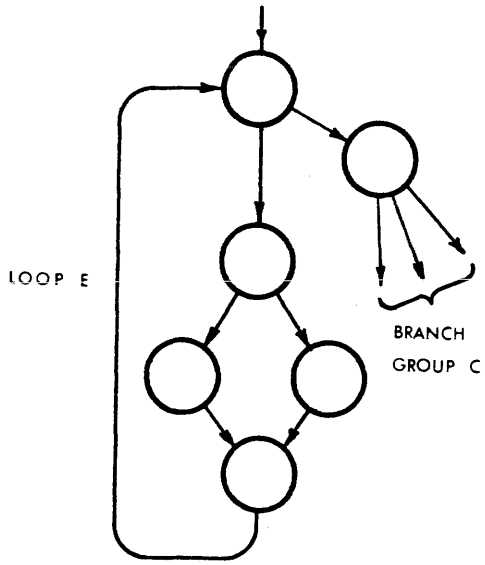


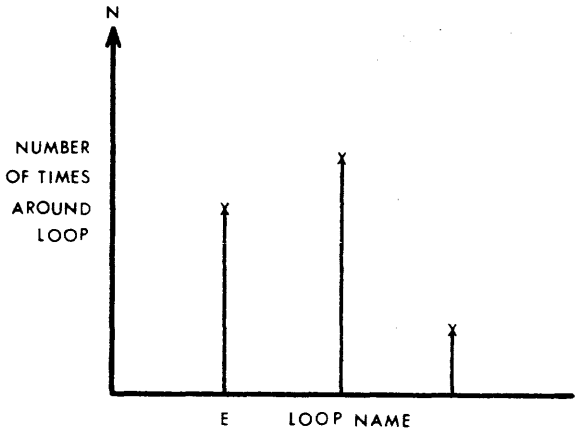
Figure 5—**Conceptual displays of program behavior**
(A) Activity during program X



(B) Busy/idle time [during program X]



(E) Graph of computation



(C) Loop factors

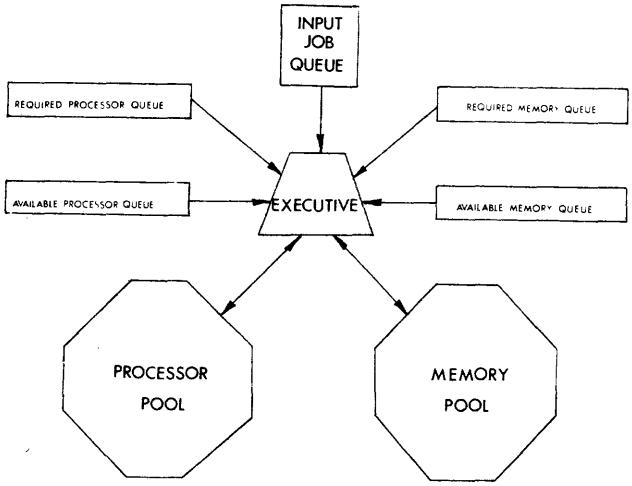
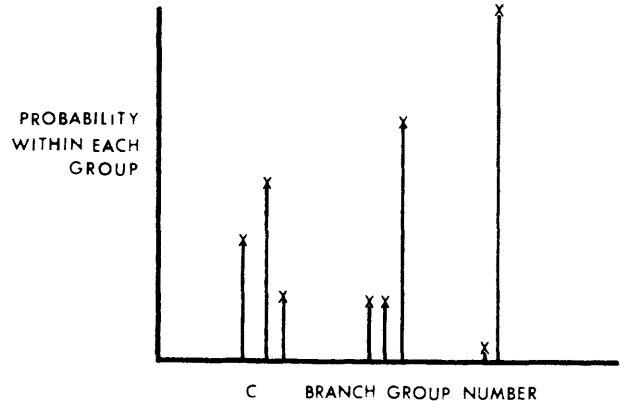


Figure 6—Model of a multiprocessor system



(D) Branch statistics

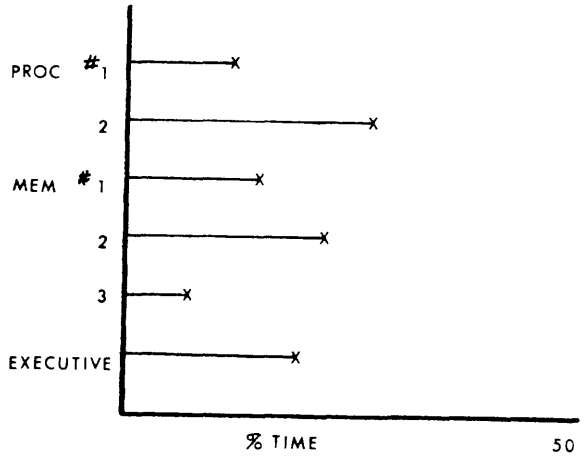
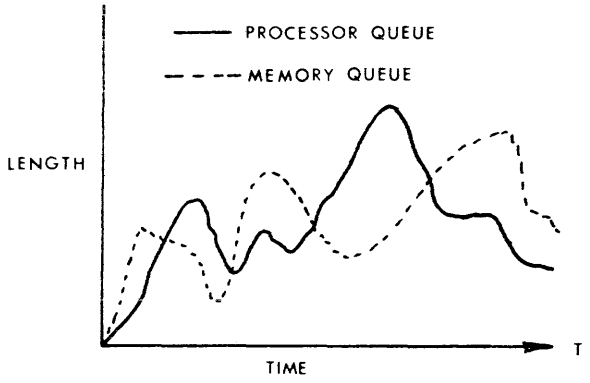


Figure 7—**Conceptual displays of multiprocessor activity**
(A) Percent utilization



(B) Queue lengths

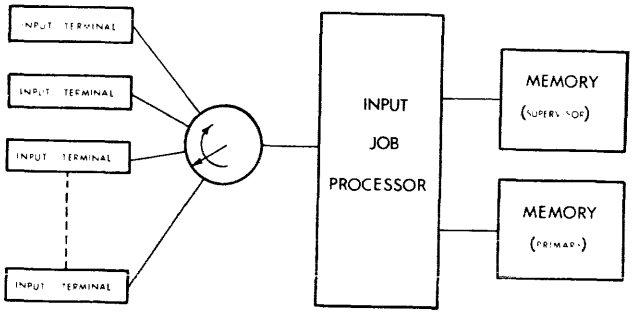


Figure 8—Model of a remote user, multiprogrammer system

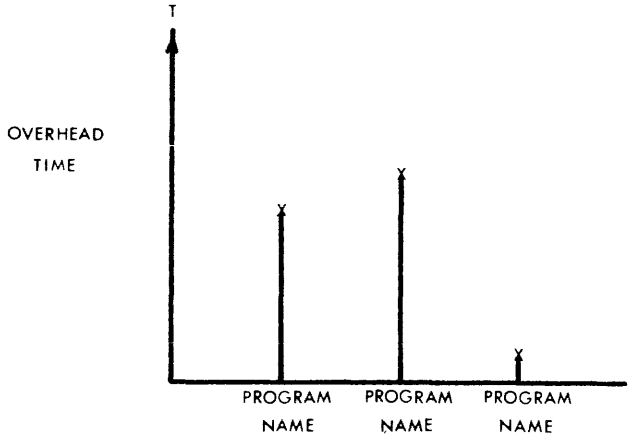


Figure 9—Conceptual display of program overhead in a multiuser, multiprogrammed system

are selected by light pen interaction and produce the results defined below:

SAVE : stores drawing image for later use.
RESET : (returns to function specification display).
ERASE : clears the screen and returns to function key mode.
ALTER : jumps to alteration table for changes in graph parameters.
MOVE : changes the screen position of the entire display image.
FIND : displays x,y coordinates of light pen interrogated points.

For example selection of the FIND function and associated light pen selection of the Arithmetic and I/O instruction groups produces the counter values in the upper right hand corner of the display.

A set of basic macroprograms required to implement such displays is listed in Appendix I.

The next part of the paper discusses the actual systems under development.

Part II - UCLA's snuper computer

The system configuration detailed in Figure 11 will be used through the several phases of development of the UCLA instrumentation automaton. As the specialized sensory system grows in complexity, improvement in capability is sought in two generally dependent directions:

1. Enlargement of the feasible class of measurements
2. Reduction of the amount of artifact.

The class of measurements towards which the design is being guided include:

1. Equipment utilization
2. Instruction type or resident routine type usage
3. Program execution activity at the statement level for source language programs
4. Program execution activity at the instruction level for machine language programs

The next part of this discussion will describe solely the third measurement (measurements one and two are easier; measurement four is harder) and consider an environment in which an object machine is used to instrument itself.

Phase 0 - self measurement

Prior to the acquisition of any special equipment, node and arc activity of FORTRAN IV programs for the IBM 7094 can be measured by the 7094 in the following manner.

Source programs will be preprocessed and each statement analyzed to find out what its possible successors are. The source program is then rewritten into a computationally equivalent source program, but which contains, at appropriate points, statements of the form

CALL EMIT (i)

where i is a unique integer. During execution, the subroutine EMIT will tally the number of times it is entered for each value of i. At the termination of execution, linear combinations of the totals yield a count of how many times each statement was ex-

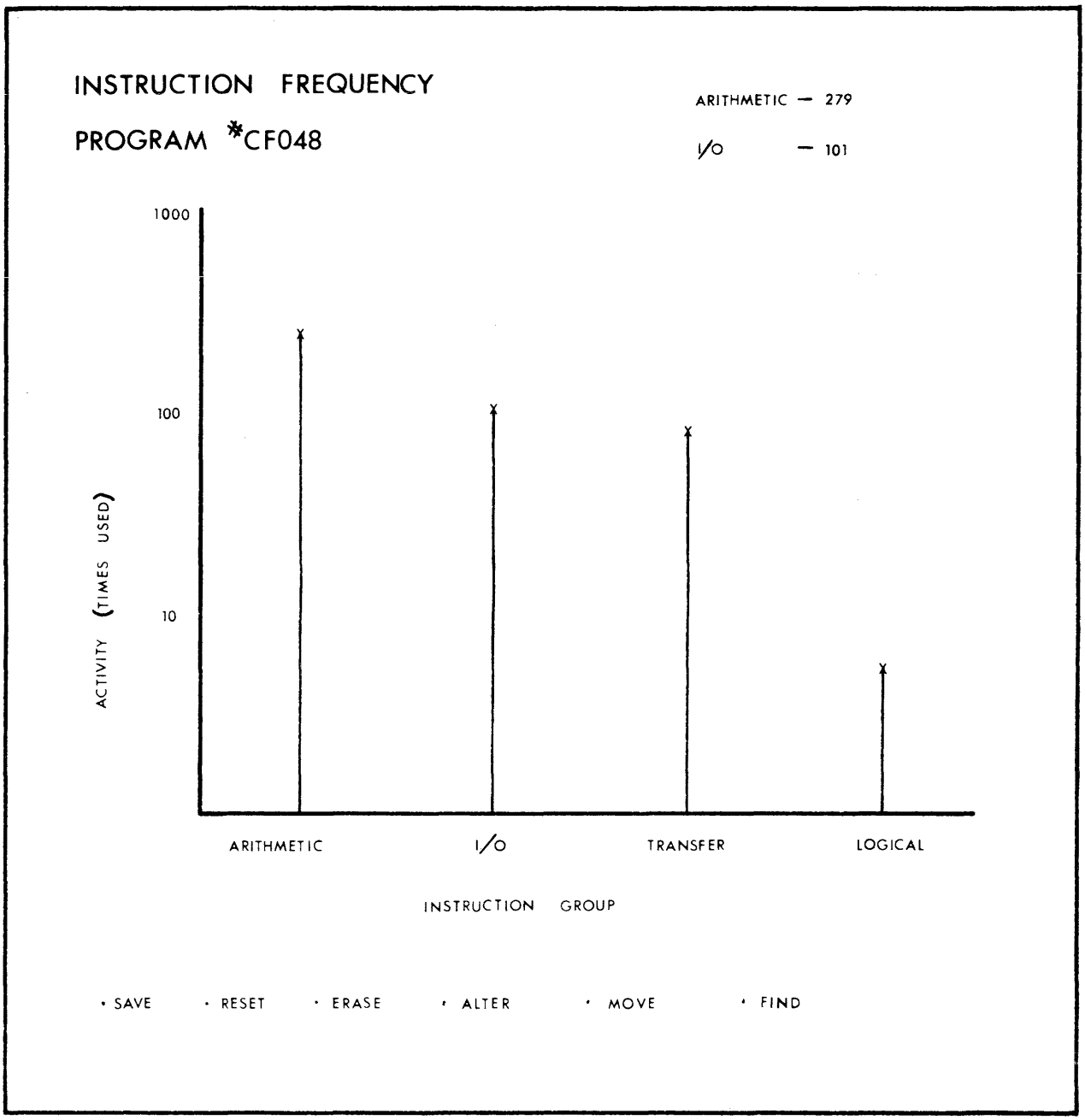


Figure 10—Example of instruction frequency display

ecuted and how many times each of the possible statement-to-statement transitions took place.

During preprocessing, heavy use is made of the fact that, with the exception of logical IF statements and the bottoms of DO loops, control leaves a statement and passes to either

1. The next sequential statement,
2. An explicitly named statement, or
3. A universal exit point.

Each statement which produces object instructions is assigned a node number; however, logical IF statements are assigned two node numbers, and a statement which terminates a DO loop is assigned an extra node number for each DO loop it terminates. The node numbers are assigned sequentially.

In the discussion of a particular statement,

The symbol	S stands for the node number assigned to the statements,
the symbols $T_1, T_2, \ldots, T_n$	stand for the node numbers assigned to those statements to which the statement may explicitly branch,

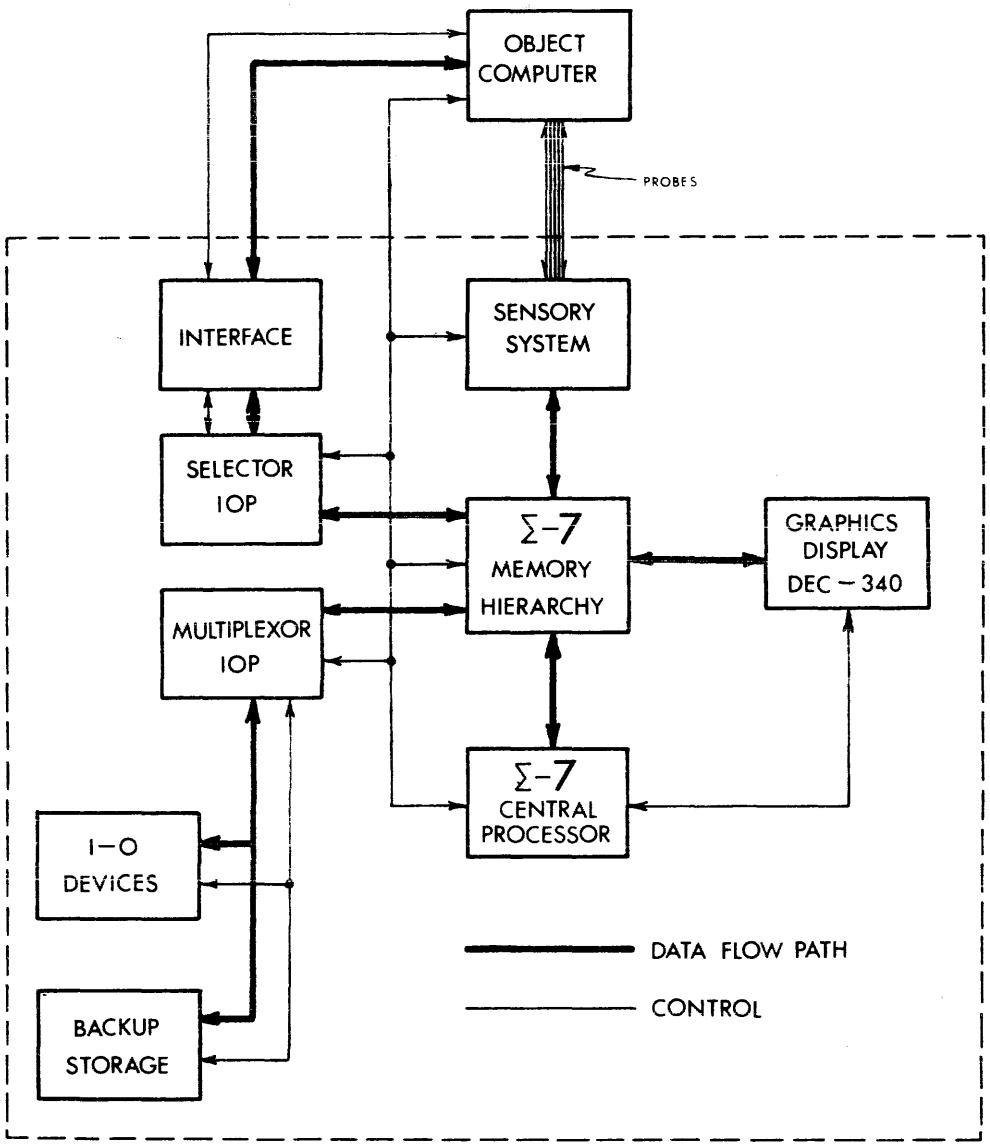


Figure 11 – System configuration for generalized instrumentation computer

the symbols $R_1, R_2, \ldots, R_m$ stand for the node numbers of those statements which may explicitly branch to the statement,

the symbol B stands for the node number of the first statement after the DO statement in a DO loop,

and the symbol E stands for the universal exit point.

The following table shows a list of the successors of a few of the FORTRAN statement types.

Statement type	**Successors**
Arithmetic Statement	S+1
Input/Output	S+1
CALL	$S+1, T_1, \ldots, T_n$
GO TO	T
Computed GO TO	$T_1, T_2, \ldots, T_n$
CONTINUE	S+1
STOP	E
Arithmetic IF	T_1, T_2, T_3
Logical IF (first half)	S+1,S+2
DO	S+1
Bottom of DO loop	B,S+1.
RETURN	E

The symbolism

(S) stands for the execution of node S,

K(S) stands for the number of times (S) occurs,

(S_1,S_2) stands for the transition of control from node S_1 to node S_2, and

$K(S_1,S_2)$ stands for the number times (S_1,S_2) occurs.

If $(P_1),(P_2),..,(P_m)$ are all of the possible immediate predecessors of (S) and (Q_1), $(Q_2),\ldots,(Q_n)$ are all of the possible immediate successors of (S), then

$$K(S)=K(P_1,S)+K(P_2,S)+...+K(P_m,S)$$
$$K(S)=K(S,Q_1)+K(S,Q_2)+..+K(S,Q_n)$$

In particular, for

Arithmetic statements, $K(S) = K(S,S+1)$; for
Logical IF statements, $K(S)=K(S,S+1)+K(S,S+2)$; for
GO TO statements $K(S)=K(S,T)$; etc.

The measurements to be made are

1. All entries into a program,
2. All explicit branching, $K(S,T_i)$, except one forward outbranching for those statements which do not have (S,S+1) outbranchings,
3. All nodes at the top of DO loops,
4. K(S+1,S+2) where S is the test part of a logical IF.

These measurements are sufficient to yield counts for all nodes and arcs.

An example of the kind of transformation the source program would be subjected to is:

```
      SUBROUTINE ABC (X,Y,Z,W,N)
      REAL X(N), Y(N), Z(N), W(N)
      DO 100 I = 1,N
IF    (X(I) .LT. 52.) CALL E7(X(I),Y(I),$70)
30    Z(I) = X(I) + Y(I)**4
      GO TO 100
70    Y(I) = X(I)/32
      GO TO 30
100   W(I) = ZERF(Z(I))
      RETURN
      END
```

The transformed program with emitter calls is shown in Appendix II.

Phase 1

In Phase 0, instrumentation is done by the object system on itself and the artifact of measurement is very high—although considerably less than would occur in a conventional interpretive mode.

Tracking the program

In Phase 1, emitter call artifact is still introduced at the statement level but external equipment captures and analyzes unique coding of events. Figure 12, illustrates the system required between the object computer and the analyzing processor. The emitted message is accepted at the interface and interpreted to be an address in the Sigma 7 memory. The sensory system causes an access to the memory location and increments a 24-bit field of the word at that address.

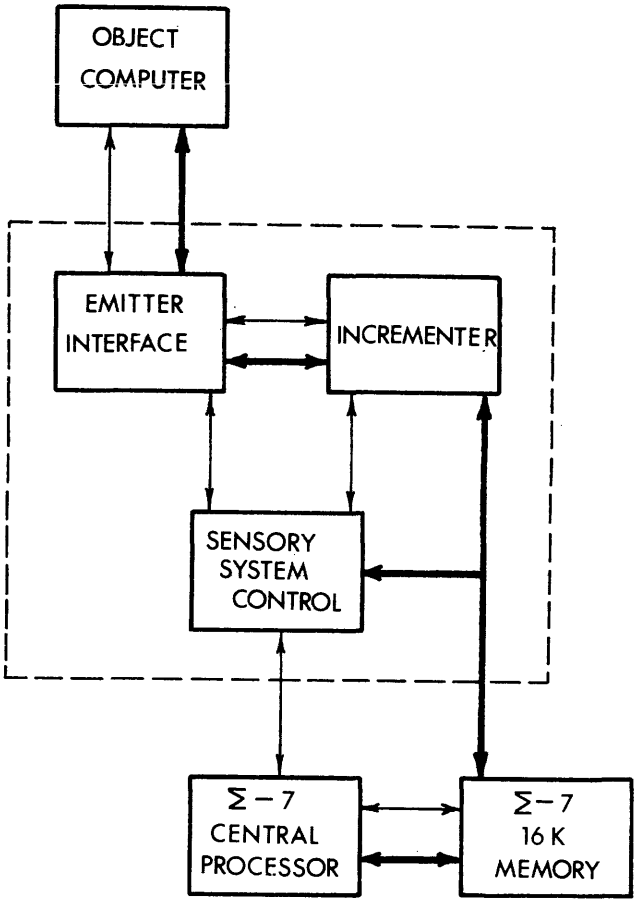


Figure 12—First phase sensory system

Thus the memory cell always contains the number of times the significant event has occurred. Between emitters, Sigma 7 programs use prior data describing the measurements to be made and the program structure to sequence appropriate generation of displays and filing of snapshot records of stored counts.

Each counter cell has an associated instruction permitting local control of the sequence of events following a recorded event. For example, the instruction can suppress further sensory system activity or can interrupt the Sigma 7 or can interrupt the object computer.

The latter case raises a point of interest. The rate at which events can be absorbed by SNUPER is limited by the time required to read, modify, and write

in Sigma 7. The capacity to absorb higher burst rates can be enhanced by high speed buffer memory in the sensory system. However there are machines whose rates of data emission are excessive. In this case there are two approaches:

1. Increase artifact by letting emitter call routines do some instrumentation in the object system;
2. Increase artifact by permitting SNUPER to interrupt the object system processing until the data is absorbed.

For data rates in the neighborhood of SNUPER's limit, instrumentation interrupts are likely to cause less artifact than emitter call routines. However as the object system speed increases, its internal routines become more effective and at some point are guaranteed to become more efficient than instrumentation interrupts.

System utilization

The Phase 0 instrumentation introduced no strong capability for measuring system utilization parameters, other than at the source language statement level.

As soon as the external equipment indicated in Figure 12 is introduced, system properties which are uniqely coded may be brought out to the emitter interface and treated in the same manner as the emitter calls. Care must be taken to capture the system states at times when they are meaningful. The next section discusses a Phase 2 development which seeks to eliminate artifact during program execution.

Phase 2 - SNUPER makes its own emitters

During the compilation and loading processes prior to execution of a program, the object system emits sufficient data about the transformed source language to let the instrumentation system prepare a map of significant object computer states for installation in the Significant Event Filter. The instrumenter has given SNUPER a formal description of the experiment and data about the object system sufficient to generate a program in the Selection System which will pass the proper class of object system states.

A block diagram of the Phase 2 system is illustrated in Figure 13. This sensory system permits observation of object computer activity without the need to insert artifact into object computer programs. The system also permits a broader class of measurements to be made.

The data selection system observes object computer clock and control signals to determine when information presented to its inputs is to be recorded in the FIFO queue. This portion of the system is programmable to permit the selection of different sets of data for observation. All data that might be of

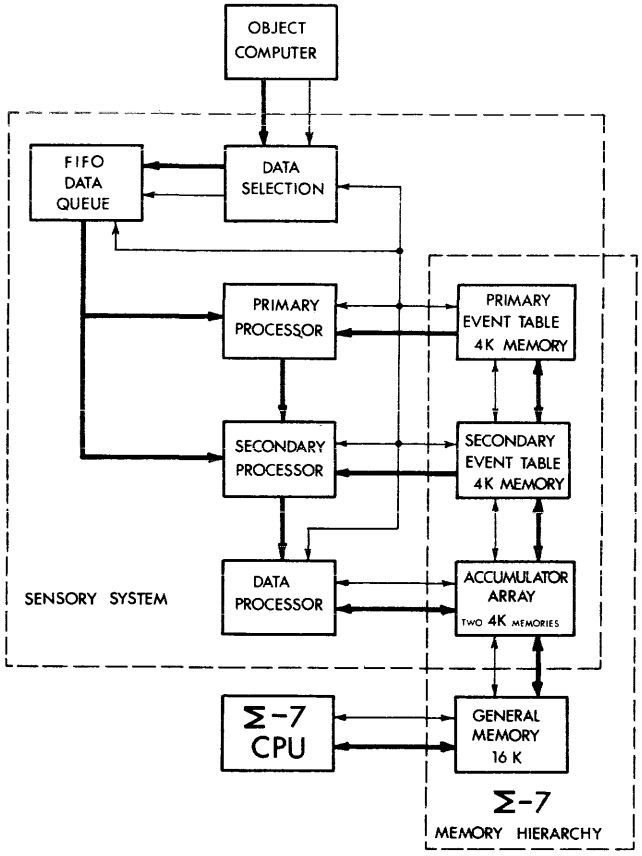


Figure 13 – Second phase sensory system

interest is placed in the FIFO queue for processing by the Primary Processor and Secondary Processor.

The function of these two processors is to filter useful information from the total set of data entering the FIFO queue. The filtered information is made available to the sensory system data processor for integration into an array of data counters. The information placed in the FIFO queue will have a format and content that is a characteristic of the particular object computer being observed. Before this data can be integrated into a data base it must be transformed into a form suitable for processing in the sensory system data processor.

The process of filtering and transforming the input stream is integrated. Each number entering the FIFO queue is used as an address for a bit in the primary event table. If the addressed bit is a one, the number is considered to be of interest. If the bit is zero, the number is considered to be valueless and it is removed from the data stream. The array of bits in the primary event table is a map of the interesting events. This map is segmented into groups that occupy part of a primary event table word. Each word contains, in addition to the map segment, a pointer that serves as the address for a word in the data accumulator array. If more than one map bit in a given primary

event table word is a one, the pointer associated with this word is modified before transmission to the sensory system data processor. The modifier is generated by locating the relative position of the selected map bit with respect to the right most non-zero map bit. The process is illustrated in Figure 14. The final step in the process is similar to that for the first phase sensory system. The selected data counter is accessed and incremented.

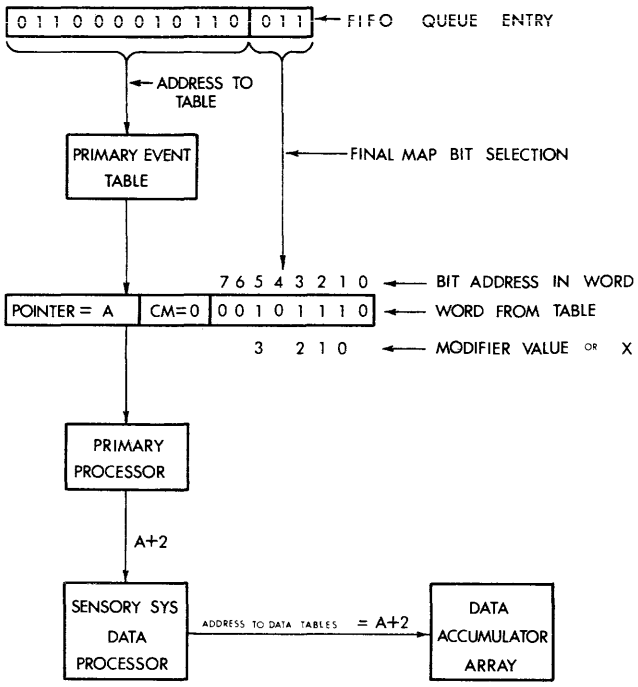


Figure 14—The filtering and map transformation processes

The operation of the Secondary Processor is identical to that of the Primary Processor. The Secondary Processor can become an extension of the Primary Processor increasing the bandpass of the sensory system, or it can become a separate processor permitting the sensory system to treat pairs of FIFO queue entries as a single event. To illustrate the latter type of operation consider the process of making observations on program arc traversals. For each instruction executed in the object computer a pair of object computer program counter values are recorded in the FIFO queue. One value specifies the location of the executed instruction in object computer memory. The second value specifies the location of the instruction defined as the successor to the executed instruction; this value must exist in the object computer when execution for the instruction in question is complete.

The primary processor operates on the first value and the secondary processor operates on the second value. If the filtering process indicates that either value is not of interest both are rejected. When both are interesting, the arc is of interest and will be processed. The transformation algorithm is similar to that for the single processor case. The major differences arise in two ways: the manner in which two modifier values are generated; and the fact that the sum of the primary processor and secondary processor modified addresses are used to select the data accumulator to be incremented. The address modifiers are generated according to the equation

$$M = x(2^{cm})$$

In this expression cm is the count multiplier value found in the selected primary event table or secondary event table word; x is the relative position of the selected map bit with respect to the right most one bit in the primary event table or secondary event table word. The location of the right most one is considered to be zero.

The hardware algorithms described above can be adapted to a variety of instrumentation experiments. The bulk of the work required to design an experiment consists of establishing software to set up the primary and secondary event tables and programming the data selection system to record the appropriate information stream. The algorithm can be implemented to have a processing rate that is limited by the read-modify-write cycle time of the memory containing the data accumulators.

CONCLUSION

This paper has focussed attention on approaches to general purpose measurement of computer system performance. Realization of such capabilities may help introduce more objectivity in system comparison. Aside from providing criteria for future system designs the study will raise consideration of changes in system organization to make them measurable to a higher degree.

Extensive experimentation is essential in order to appropriately separate classes of instrumentation which cannot be effected with reasonable equipment from classes of instrumentation which can give significant measures of system performance through a SNUPER COMPUTER of low enough cost and size that it can move from one installation to another.

REFERENCES

1 C CONTI
System aspects:System/360 model 92
Fall Joint Computer Conference Proceedings
American Federation of Information Processing Societies
Spartan Books Washington D C Vol 26 Pt II p 81 1964

2 M GRAHAM
Private Communication
University of California Berkeley

3 G H FINE C W JACKSON and P V McISAAC
Dynamic Program Behavior Under Paging
System Development Corporation SP-2397 June 16 1966

4 J I SCHWARTZ E G COFFMAN and C WEISSMAN
A General Purpose Time-Sharing System
Spring Joint Computer Conference Proceedings
American Federation of Information Processing Societies
Spartan Books Washington D C 1964 Vol 25 pp 397-411
5 E G COFFMAN B KRISHNAMOORTHI
Preliminary analyses of time-shared computer operation
System Development Corporation SP1719 August 1964
6 B KRISHNAMOORTHI R C WOOD
Time-shared computer operations with both inter-arrival and service times exponential
System Development Corporation SP1848 October 1964
7 R A TOTSCHEK
An empirical investigation into the behaviour of the SDC time sharing system
Systems Development Corporation SP2191 August 1965
8 A L SCHERR
An analysis of time-shared computer systems,
PhD Dissertation Department of Electrical Engineering
Massachusetts Institute of Technology
Cambridge Massachusetts June 1965
9 T G STOCKHAM JR
Some methods of graphical debugging
IBM Scientific Computing symposium
Man-Machine Communications
Thomas J Watson Research Center
Yorktown Heights New York May 3-5 1965
10 B L WOLMAN
A subroutine trace program for CTSS
Massachusetts Institute of Technology
Project MAC MAC M168 2 March 5 1965
11 S WHITELAW M JONES
Statistics for CTSS
Massachusetts Institute of Technology
Project MAC MAC M 161 June 2 1964
12 W BUCHHOLZ
Planning a computer system
McGraw Hill Book Co.
New York 1962 p 21 p 228
13 D MARTIN G ESTRIN
Experiments on Models of computations and systems
IEEE Trans on Electronic Computers in Press
14 G M AMDAHL
IBM Corporation
San Jose California private communication
15 P CALINGAERT
System Performance evaluation survey and appraisal
Comm of the ACM Vol 10 No 1 12-18 January 1967

APPENDIX I

Basic macros required for instruction frequency display

Specification of data file containing instruction count
Specification of double word format
Specification of coordinate system
Specification of vertical range
Specification of number of horizontal increments or cycles
Specification of coordinate labels (instruction types)
Specification of conversion transform
Display of counter values of interrogated instructions
Intensification of interrogated instructions
Display and alteration of counters and registers

APPENDIX II

```
      SUBROUTINE ABC(X,Y,Z,W,N)
      REAL X(N), Y(N), Z(N), W(N)
      CALL EMIT (1)
      DO 1 I = 1,N
      CALL EMIT (2)
      IF (.NOT.(X(I).LT.52.)) GO TO 30
      CALL E7(X(I),Y(I),$2)
      CALL EMIT (4)
30    Z(I) = X(I) + Y(I)**4
      GO TO 100
70    Y(I) = X(I)/32.
      GO TO 3
100   W(I) = ZERF (Z(I))
      GO TO 1
2     CALL EMIT (3)
      GO TO 70
3     CALL EMIT (5)
      GO TO 30
1     CONTINUE
      RETURN
      END
```

An algorithmic search procedure for program generation

by M. H. HALSTEAD, G. T. UBER, and K. R. GIELOW
Lockheed Missiles and Space Company
Sunnyvale, California

INTRODUCTION

A programmer in writing and checking out an arithmetic computer program provides both a procedural description of the program and a set of numeric test cases. Similarly, a student of elementary physics is often given a set of formulae to be used, and correct answers to problems. These two inputs in both cases provide a redundancy which gives some measure of confidence that the procedure is correct. In theory either is adequate, subject to the limitations of induction. While compilers operate upon the first type of data, several programs have been written which use the second type.

Friedberg[1,2] in 1957 attempted to generate programs to compute Boolean functions in his experiments on machine learning. The impression which his experiments left was that a technique which generated and tested programs showed little promise. More recently Simon tackled the problem of generating IPL-V programs with his Heuristic Compiler,[3] which uses means-end analysis of the before and after states of IPL-V stacks.

The procedure described in this paper searches for programs which compute arithmetic functions. Its input is one or more data sets each composed of one or more numeric arguments called parameters and a numeric function value. The exhaustive forward directed search—named the "British Museum Algorithm" by Newell, Shaw, and Simon[4]—was chosen because of its simplicity and because of the facility of computers for executing arithmetic programs. The search is optimized through utilization of syntactic symmetry as defined by Gelernter[5] to eliminate redundant programs. Where applicable, the physical dimensions of the data are also used.

The search program described in the remainder of this paper establishes a benchmark for the exhaustive search approach. A basic search rate of 7,000 trial programs per second is achieved, which is a hundredfold improvement over Friedberg's results. The program represents an application of computers to that oft forgotten subject in artificial intelligence: the numeric problem.

An experimental search program

A computer program named MAGIC has been written for the IBM 7094 to execute the search. For speed the program is written in NELIAC and FAP rather than in an interpretive language such as IPL-V. It now consists of some 16,000 words of instructions and tables.

Program generation

The goal of the search program is to find the shortest sequence of program steps which computes a function to within a specified tolerance. A program step is an operator-operand pair such as (ADD A). All possible sequences of program steps are generated and executed, and the results are tested against the function value. To ensure finding the shortest program, the search program first generates all one-step programs

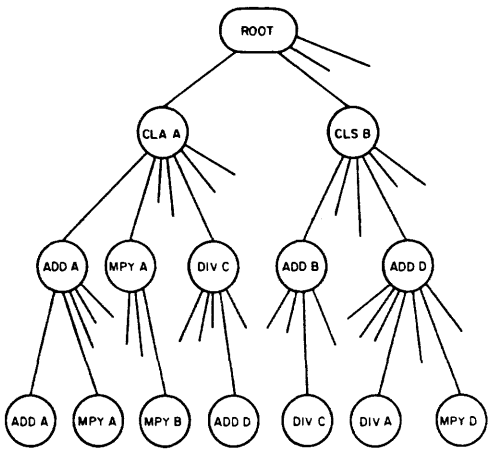


Figure 1—Partial program tree for three steps

SUCCEEDING OPERATION	CURRENT OPERATION: 0+ ABCDE	0- ABCDE	+ ABCDET	- ABCDET	* ABCDET	/ ABCDET	$a \rightarrow T$	$a \rightleftarrows T$	$\frac{1}{a}$	sin a	cos a	$\sqrt{a}$
0+ A	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0+ B	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0+ C	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0+ D	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0+ E	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0- A	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0- B	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0- C	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0- D	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
0- E	-----	-----	-----	-----	-----	-----	1	-	-	-	-	-
+ A	10000	00000	100000	-00000	111111	111111	1	1	1	1	1	1
+ B	11000	00000	110000	1-0000	111111	111111	1	1	1	1	1	1
+ C	11100	00000	111000	11-000	111111	111111	1	1	1	1	1	1
+ D	11110	00000	111100	111-00	111111	111111	1	1	1	1	1	1
+ E	11111	00000	111110	1111-0	111111	111111	1	1	1	1	1	1
+ T	-----	-----	111111	11111-	111111	111111	1	1	1	1	1	1
- A	-1111	10000	-00000	100000	111111	111111	1	1	1	1	1	1
- B	1-111	11000	1-0000	110000	111111	111111	1	1	1	1	1	1
- C	11-11	11100	11-000	111000	111111	111111	1	1	1	1	1	1
- D	111-1	11110	111-00	111100	111111	111111	1	1	1	1	1	1
- E	1111-	11111	1111-0	111110	111111	111111	1	1	1	1	1	1
- T	-----	-----	11111-	111111	111111	111111	-	1	-	1	1	1
* A	10000	10000	111111	111111	100000	-00000	1	1	-	1	1	1
* B	11000	11000	111111	111111	110000	1-0000	1	1	-	1	1	1
* C	11100	11100	111111	111111	111000	11-000	1	1	-	1	1	1
* D	11110	11110	111111	111111	111100	111-00	1	1	-	1	1	1
* E	11111	11111	111111	111111	111110	1111-0	1	1	-	1	1	1
* T	-----	-----	111111	111111	111111	11111-	1	1	-	1	1	1
/ A	-1111	-1111	111111	111111	-00000	100000	1	1	-	1	1	1
/ B	1-111	1-111	111111	111111	1-0000	110000	1	1	-	1	1	1
/ C	11-11	11-11	111111	111111	11-000	111000	1	1	-	1	1	1
/ D	111-1	111-1	111111	111111	111-00	111100	1	1	-	1	1	1
/ E	1111-	1111-	111111	111111	1111-0	111110	1	1	-	1	1	1
/ T	11111	11111	111111	111111	11111-	111111	-	1	-	1	1	1
$a \rightarrow T$	-----	-----	111111	111111	111111	111111	-	-	1	1	1	1
$a \rightleftarrows T$	-----	-----	111111	111111	111111	111111	-	-	1	1	1	1
1/a	11111	11111	111111	111111	111111	-----1	1	1	-	1	1	1
sin a	11111	11111	111111	111111	111111	111111	1	1	1	1	1	1
cos a	11111	11111	111111	111111	111111	111111	1	1	1	1	1	1
$\sqrt{a}$	11111	11111	111111	111111	111111	111111	1	1	1	1	1	1

Note: The operands A, B, C, D and E are parameters: T is a temporary storage cell, and a is the intermediate result corresponding to the contents of the IBM 7094 AC-register. The first seven operators correspond to the 7094 commands: Clear and Add, Clear and Subtract, Floating Add, Floating Substract, Floating Multiply and Proceed, Floating Divide and Proceed, and Store. The next operator exchanges the AC-register with T, and the remaining operators replace a with the indicated function of a. On the first step only the 0+ and 0- operators may be executed.

Figure 2—Successor tables

and then successively generates longer programs until a successful program is found.

Tree searching

The program space may be considered a tree in which each program is represented by a path from the root to a leaf, or terminal node. (See Figure 1.) The paths are generated in Iverson's[6] left list matrix order so that intermediate results may be saved and used without recomputation. The time required to execute the second program (CLA A, ADD A, MYP A) is thus the time needed to retrieve the result of the second step and to execute the new third step MPY A.

Successor selection

Successor criteria shorten the search by pruning subtrees rooted at each node. They are implemented with Successor Tables (Figure 2) which define the legal successors (marked "1") to each operation. While these tables bear a superficial resemblance to CO-NO tables as employed in table-driven compilers,[7] they are inherently quite different. Operations (marked "-") which cancel previous operations (e.g., SUBTRACT A and ADD A) are prohibited. Operations marked "0" are suppressed because they result in computationally equivalent sequences due to commutivity (e.g., ADD A, ADD B versus ADD B, ADD A).

Operations violating certain physical constraints are eliminated also. Investigation of a sequence is terminated if floating point overflow or underflow occurs or if division by zero is attempted. The search is also terminated if the choice of operation results

in a dimensional inconsistency (e.g., apples may not be added to oranges). The dimensionality of the intermediate results is computed, and the operator choice is tested against the following rules:

1. For addition and subtraction the intermediate result and the parameter must have the same dimension.
2. For square root the dimensions must be perfect squares.*
3. For trigonometric functions, the intermediate result (i.e., the argument) must be dimension-

Because there can be no successful program shorter than the shortest dimensionally correct program, only those operations which affect dimensions are tried until the first such program is found.

The successor tables are defined by macro instructions and are executed using a combination of double indirect addressing and triple indexing. The operators are also defined by macros and are expanded for each argument. Coding is included in the operator definitions which tests the dimensions and computes the dimensions of the result. The six dimensions are each represented as six-bit fields, the exponents having a range of ±15. The numerical values are stored as floating point numbers.

Test routine

When the last step of a sequence has been generated and executed, the result is checked against the desired result and the check counter incremented. If the dimensions agree and the numerical value is within a specified tolerance for the current function value, the successful sequence is saved in an encoded form for further processing.

Timing

Most of the search time is spent in processing the terminal nodes. It requires 60 microseconds to generate and execute the terminal step and 60 microseconds to check the final result and increment the check counter. The search is rapid enough so that 10 percent of the machine time is required merely to tally the number of sequences checked.

Search length

Given P parameters and R operators in the repertoire at each step, the maximum number of possible operator-parameter pairs is the product PR. With β pairs at each step, the number of possible sequences of length S is β^S, or as a maximum $(PR)^S$. With the British Museum Algorithm, sequences are examined in order of length. To find a sequence of length S requires Q checks where

$$Q = \sum_{i=1}^{S} \beta^i$$

Twelve operators and five parameters are sufficient for the problems being investigated; hence, β equals 60. Not all operators need operate upon all parameters. Let P be the number of input parameters, R_p the numbers of operators acting upon input parameters, T the number of temporary storage cells, R_T the number of operators acting upon such cells, and F the number of operators (Functions) acting upon intermediate results. We now define

$$\beta = PR_P + TR_T + F$$

For the present Successor Tables: P is 5, R_P is 12; T is 1; R_T is 6, and F is 4. Hence β is 40, yielding the curve "β=40" in Figure 3. The other constraints in the Successor Tables reduce the value of β still further. For undimensioned parameters (NO DIMENSIONS) the effective value is 18. When the parameters are dimensioned such that addition and subtraction are completely suppressed (COMPLETE DIMENSIONS), the value drops to 3. However, for a typically dimensioned problem β has a value of about 6. The curves are computed for five parameters, and the points in the figure are labeled with the search time in seconds.

Examples

The search timing for most problems can be predicted from Figure 3. The examples below indicate typical results and also illustrate additional features of the system.

Multiple data sets provide a better basis for generalization than do single data sets.* To avoid redundant printout, programs are listed only if they satisfy all data sets or else satisfy a combination of data sets not included in combinations satisfied by previously listed programs.

The results of a search over multiple data sets are expressed by the list of programs and by a Boolean cover matrix C in which each element $c_{p,d}$ equals 1 if the program p computes an acceptable value for the data set d, and equals 0 otherwise. The decision

*This constraint was introduced because the current representation of dimensions does not permit fractional powers.

*Rigg's[8] advice to biologists seems appropriate: "You should check any procedure about which you are dubious by substituting sensible numerical values for the algebraic terms and making sure that the procedure is correct for these numbers. To guard against the possibility that a wrong procedure may by chance give a correct result with the particular numbers chosen, you should check by this method of numerical substitution twice with two different sets of values."

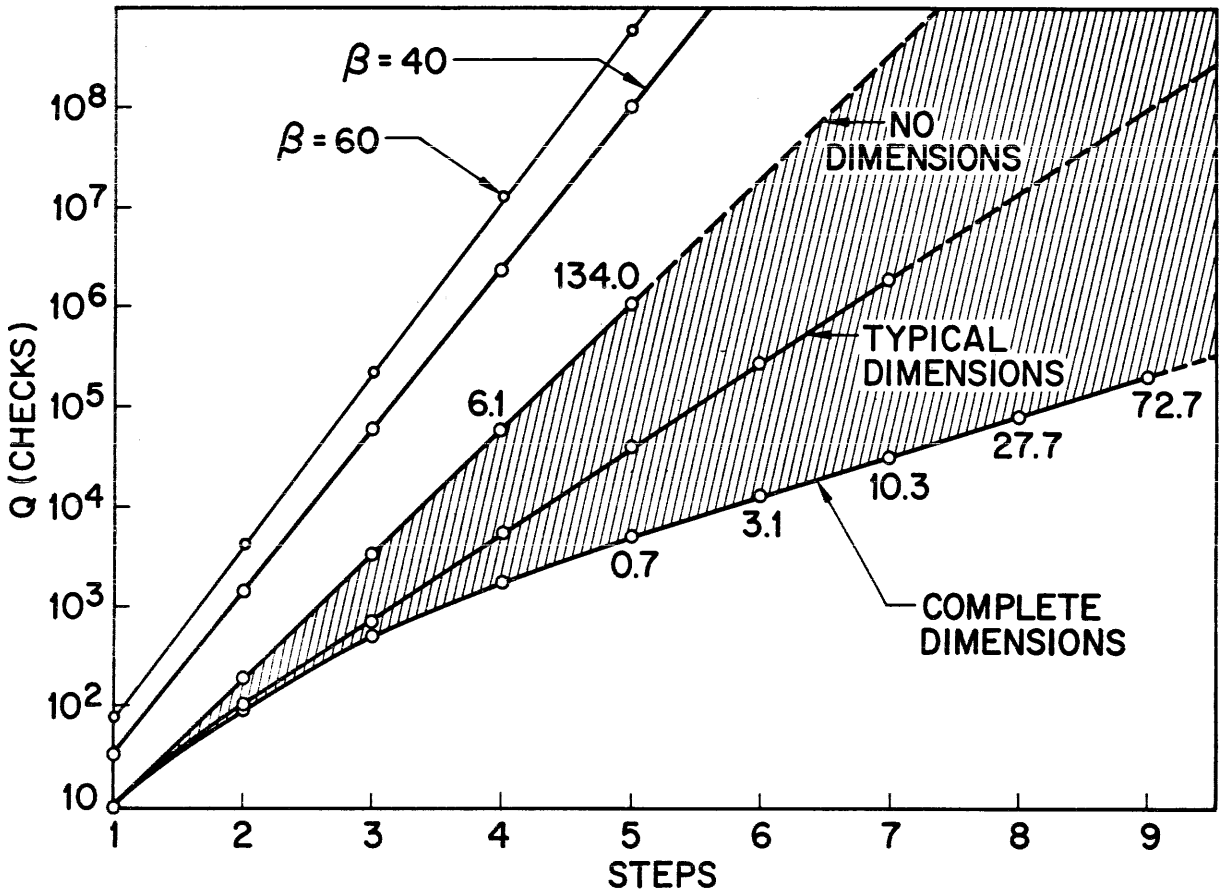


Figure 3 – Search times

program generator, operating upon C and the data set parameters, searches for a minimal set of programs p each with an associated Boolean function b_p such that:

(1) for each data set d there exists a program p such that $b_p(d) = 1$
(2) $b_p(d)$ implies $c_{p,d}$
(3) b_p is a Boolean expression of threshold functions of single parameters

Example one: approximating functions

The tolerance was set to 10 percent so that approximate solutions would be listed. The search, using input shown in Table I, resulted in the programs of Table II and required 3.1 seconds.

Table I

INPUT DATA FOR APPROXIMATION EXAMPLE

Name Dimension	Parameter				
	P	T (deg)	v_o (m/sec)	T_o (deg)	v (m/sec)
Data Set					
1	1.0	0	331.7	273	331.7
2	1.0	20	331.7	273	344.0
3	1.0	100	331.7	273	386.0
4	1.0	500	331.7	273	553.0
5	1.0	1000	331.7	273	700.0

The expression $(1/\sin P)^{1/2} = 1.09$ is in effect a coefficient used to "fit" the function to the first three data sets. Coefficients (for example, the parameters v_o and T_o) must usually be supplied while the functions are generated, in contrast to most curve fitting programs for which the functions are supplied and the coefficients are generated.

Table II

OUTPUT FOR APPROXIMATION EXAMPLE

Program	Function	Checks	Cover Matrix Data Sets 1 2 3 4 5
1	$v = v_o$	5	1 1 0 0 0
2	$v = (1/\sin p)^{1/2} v_o$	7,464	1 1 1 0 0
3	$v = [(T + T_o)/T_o]^{1/2} v_o$	16,663	1 1 1 1 1

Example two: decision program

This shows a piecewise fit to a payroll function. The input data for two employees are given in Table III. The search, using a tolerance of 1 percent, found 780 programs of which the 8 in Table IV were listed.

Table III

INPUT DATA FOR DECISION PROGRAM EXAMPLE

Name Dimension	Hours Worked (hr)	Parameter			Function		
		Day of Week	Hourly Rate ($/hr)	Shift Length (hr)	Constant	Days Pay ($)	
Data Set							
0	7.6	1.0	2.75	8.0	2.0	20.90	
1	8.0	2.0	2.75	8.0	2.0	22.00	
2	9.3	3.0	2.75	8.0	2.0	27.36	
3	8.0	4.0	2.75	8.0	2.0	22.00	Employee 1
4	5.3	5.0	2.75	8.0	2.0	14.58	
5	10.7	6.0	2.75	8.0	2.0	33.14	
6	9.0	7.0	2.75	8.0	2.0	49.50	
7	10.4	1.0	3.60	8.0	2.0	41.76	
8	6.0	2.0	3.60	8.0	2.0	21.60	
9	8.0	3.0	3.60	8.0	2.0	28.80	
10	4.0	4.0	3.60	8.0	2.0	14.40	Employee 2
11	11.2	5.0	3.60	8.0	2.0	46.08	
12	6.3	6.0	3.60	8.0	2.0	22.68	
13	6.0	7.0	3.60	8.0	2.0	43.20	

Table IV

COVER MATRIX FOR DECISION PROGRAM EXAMPLE

Program \ Data Set	0	1	2	3	4	5	6	7	8	9	10	11	12	13	Function
1	1	1	0	1	1	0	0	0	1	1	1	0	1	0	p = hr
2	0	1	1	1	0	1	0	1	0	1	0	1	0	0	p = ((h - s)/2.0 + h)r
3	0	1	0	1	0	0	0	0	0	1	0	0	0	1	p = (2.0(s - h) + s)r
4	0	0	1	0	0	0	0	0	0	1	1	0	0	0	p = (hd - s)r/2.0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	1	p = (r/2.0 + r)s
6	0	0	0	0	0	0	1	0	0	0	0	0	0	1	p = (h + h)r
7	0	0	0	0	0	0	1	0	0	1	0	0	0	0	p = $\sqrt{d - 2.0}$ rs
8	0	0	0	0	0	0	0	0	0	0	0	0	1	1	p = (h sin d + s)r

During 3.26 minutes of computer time, 1,311,126 programs were tested.

The decision program generation required 0.02 minutes and 68 trials. The generator combined three programs (in NELIAC[7]) as shown in Figure 4. The composite program has 16 steps although the maximum search length was five steps.

Example three: program library

This example illustrates the construction of a specialized subprogram library. For each of the six blackbody radiation problems, the resulting programs in Table V were stored for use on the remaining problems. A stored subprogram can be executed as a step, its input being a permutation of the input parameters and the contents of T. The program for Function 5, for example, consists of

(1) Function 4(A, B, C, D)
(2) DVP E
(3) DVP E

where parameters A, B, C, D and E correspond to the variables r, e, s, T, and d, respectively. While the maximum search time for a given number of steps is roughly proportional to library size, a properly specialized library could significantly shorten the search depth.

DISCUSSION

The principal object of this paper has been to investigate the forward directed search as implemented on present-day computers. While improvements in computer performance would be expected to lower the search cost still further, the greatest reductions may be expected from the use of matching and selection procedures which extract operators, parameters, and subprograms most applicable to a given problem and supply them to the search procedure.

REFERENCES

1 R M FRIEDBERG
A learning machine part I

```
HOURS WORKED   =  0.0,
DAYS OF WEEK   =  0.0,
HOURLY RATE    =  0.0,
SHIFT LENGTH   =  8.0,
TWO            =  2.0, ;
COMPUTE DAYS PAY:
    {IF DAY OF WEEK  =  7.0:
        (HOURS WORKED + HOURS WORKED) × HOURLY
            RATE → DAYS PAY;
     IF NOT,
        IF HOURS WORKED ≤ 8.0:
            HOURS WORKED × HOURLY RATE → DAYS PAY;
        IF NOT,
            ((HOURS WORKED − SHIFT LENGTH)/TWO
            + HOURS WORKED) × HOURLY RATE
            → DAYS PAY;;}
```

Figure 4—Output for decision program

Table V

PROBLEM SEQUENCE USING LIBRARY FUNCTIONS

Function		Steps	Checks	Total Steps
1	sT^4	5	94	5
2	esT^4	2	58	6
3	$AesT^4$	2	237	7
4	r^2esT^4	2	1259	8
5	r^2esT^4/d^2	3	17296	10
6	$AsT^4/d^2\pi$	4	155789	9

IBM Journal of Research and Development vol 2 pp 2-13 Jan 1958

2 R M FRIEDBERG B DUNHAM J H NORTH
A learning machine part II
IBM Journal of Research and Development vol 3 pp282-7 Jul 1959

3 HERBERT A SIMON
The heuristic compiler
Memorandum RM-3588-PR The RAND Corporation Santa Monica Calif 1963

4 A NEWELL J C SHAW H A SIMON
Empirical explorations of the logic theory machine
Proceedings of the Western Joint Computer Conference pp 218-39 1957
Reprinted in Feigenbaum E A and J Feldman
Computers and thought
McGraw-Hill Book Company Inc pp 109-33
New York 1963

5 H GELERNTER
A note on syntactic symmetry and the manipulation of formal systems by machine
Information and control
vol 2 pp 80-89 Apr 1959

6 K E IVERSON
A programming language
John Wiley & Sons Inc New York 1962

7 M H HALSTEAD
Machine-independent computer programming
Spartan Books Washington D C 1962

8 D S RIGGS
The mathematical approach to physiological problems
Williams & Wilkins Baltimore p 3 1963

A macro—generator for ALGOL

by H. LEROY

Compagnie Bull—General Electric
Paris, France

INTRODUCTION

The concept of macro-facility is ambiguous, when applied to higher level languages.

For some authors[1,2] a macro-facility is essentially a means to define, inside a program, local extensions to the language in which the program is written. It is usually understood that these definitions are interpreted before execution, but this is an implementation feature rather than a language feature, and, in this sense of the concept of macro-facility, the procedure mechanism of ALGOL must logically be regarded as a macro-facility.

In the macro-generator defined in this paper, there is no facility for defining new linguistic entities in ALGOL. It is a mechanism of generation of ALGOL programs, programmed in a special language, macro-ALGOL, and the fact that an ALGOL program is completely generated before being executed is an essential linguistic feature of macro-ALGOL.

The motivation here is not, therefore, the comfort of the programmer. The macro-generator is intended to be useful in situations where the production of an ALGOL program by human programming power would be too expensive or nearly impossible, or where its mechanical production would mean a substantial reduction of cost. Examples of such situations are:

- The production of a very big and complex program with relatively simple specifications, the complexity of the program arising from the fact that conceptually distinct specifications must be inextricably mixed up in the program.
- The production of many programs with small differences in their specifications, these differences implying, however, substantial differences in their structures to preserve efficiency.
- In heuristic proramming, where many algorithms must be tested before one is found whose behavior is satisfactory. In this case, the ease of modification can only be achieved if each version of the algorithm can be programmed mechanically from a convenient specification.

Our system is therefore intended to be a system of computer aided ALGOL programming, and its expected advantages are those of any mechanization. Just as a program is better executed by a machine than by a man, the production of a program or a set of programs can involve a lot of repetitive thinking, and that part of the programming work can best be done by a machine.

Characteristics

It is clear that, whatever the system used, the programmer who instructs a macro-generator to produce a program must think at a level higher in abstraction, at least, than when he writes a program with his own hand. Macro-programming is therefore unavoidably more difficult than direct programming. It is suggested that a macro-generator cannot be significantly useful if it is not very powerful and flexible. For this to be achieved, we assume that the following requirements must be met, at least.

1. Full power of the macro-generator in general purpose information processing

This requirement might look strange at first sight. It is quite natural, on the contrary: for example, the generation of a polynomial approximation requires the computation of the numerical coefficients, and maybe the degree, of the polynomial, and this computation may be of any complexity.

We will give this requirement a more precise form: a generator of ALGOL programs must be able to do anything that can be programmed in ALGOL.

This immediately suggests that macro-ALGOL, the programming language of the macro-generator, must be a true extension of ALGOL.

2. Conceptual enconomy in the definition of the generating mechanism

In other words: the extension that makes ALGOL into macro-ALGOL must be defined with as few ad-

ditional notions as possible. This means that macro-ALGOL must be a natural and straightforward extension of ALGOL or, more precisely, that most generating mechanisms of macro-ALGOL be natural and straightforward generalizations of the mechanisms already present in ALGOL.

3. Syntax independance

The programmer is interested in the semantical contents of programs, and not in their representations as strings of basic symbols. This implies that the macro-generator must not be a generator of arbitrary strings of basic symbols. The ability of generating strings which would not conform to the ALGOL syntax can be of no use, and if it did exist, then the macro-programmer would be entirely responsible for the generated string obeying the rules of ALGOL syntax. Since these rules are defined once and for all, the macro-generator can, and therefore must take care of them.

It is not even necessary to assume that the generated program will always appear in the form of a string in the ALGOL syntax. If it must be immediately interpreted by a mechanism, then another form would probably be preferable.

The operation of the macro-generator can therefore be defined in terms of the abstract structure of the generated program. If this abstract structure must be translated into a string, then the implementor of the macro-generator, and not the macro-programmer, is left the choice of one string among the strings which represent this structure.

Furthermore, the implementor is left free to take advantage of obvious cases of semantical equivalence: for example, he can detect and erase all unlabelled dummy statements, replace 'a+O' by 'a' and '-(-a)' by 'a'.

The idea of syntax independence has been inspired by the paper of P. J. Landin.[3]

4. Mechanical generation of macro-programs

It may be necessary to perform the generation of an ALGOL program in several successive steps. This is possible if the macro-generator can generate any program in macro-ALGOL, that is in its own language. We can now concentrate on macro-ALGOL for itself, as a programming language in its own right, with a property of macro-generation. We can even define a macro-language, without reference to any existing language, thus:

'A macro-language is a programming language such that any program in this language, when executed, possesses a value, which is a program in the same language.'

'A macro-generator is the processor of some macro-language.'

(The value of a program is obviously what we called the generated program.)

Macro-ALGOL contains a distinguished subset, namely ALGOL. This subset has the following property:

'The value of a program in ALGOL is always equivalent to a dummy statement, i.e., a program whose execution has no effect, besides producing another dummy statement as value.'

We can now envision the generation of an ALGOL program, say in 2 steps:

- A program is written, by hand, in macro-ALGOL.
- This program is executed by the processor of macro-ALGOL (the ALGOL macro-generator), with a set of input data. This yields various results, known as output data, and a distinguished output, which is the value of the executed program.
- The value just obtained is executed, as a program, by the macro-ALGOL processor, with another set of input data, which yields another value, which turns out to be a program in ALGOL.
- The value just obtained is executed, as a program, by the macro-ALGOL processor. The value obtained is a dummy statement.

Further executions would have no other effect than producing dummy statements as values.

5. Efficient implementation

It must be pointed out that, until now, no assumption has been made on the implementation of macro-ALGOL. We have considered the macro-generator as an ideal machine whose machine language is macro-ALGOL. However, any practical implementation of macro-ALGOL will make use of another machine, with a compiler of macro-ALGOL built for this machine. Therefore, what we called the execution of a program in macro-ALGOL will take place in two steps:

- compilation of the program, i.e., its translation into the language of a machine M;
- execution, by machine M, of the result of the translation. This execution involves, in general, ordinary input-output activities, and always the output of a program in macro-ALGOL.

Furthermore, this implementation must be efficient. More precisely:

'The object program produced by the compilation of an ALGOL program by the macro-ALGOL compiler is as efficient as the object program produced, for the same machine, by an ordinary ALGOL compiler.'

Outline

The extension that makes ALGOL into macro-ALGOL is the introduction of a new class of identifiers, namely *symbolic identifiers*. Whether an identifier is symbolic or not is determined by its declaration.

A non symbolic identifier is treated exactly like in ALGOL.

A symbolic identifier is treated in the following way:

If it is a variable identifier, the variable is never assigned a value. When it occurs as the left part of an assignment statement, the assignment is not performed, but an assignment statement is produced, with the same identifier as left part, and put in the generated program. If it occurs in an expression, it appears in an expression which is produced as value of this expression, and put in the generated program.

For example, if n is a symbolic identifier, the statement

$$n := n + 1$$

has no other effect than producing the statement

$$n := n + 1$$

If a is an ordinary variable whose current value is 5, the statement:

$$n := n + a$$

produces the statement

$$n := n + 5;$$

If it is a label identifier, a go to pointing to it is not executed and produces a go to pointing to the same label;

If it is a procedure identifier, any call to this procedure produces another call to the same procedure.

The declaration of a symbolic identifier produces a declaration of an identifier, which may again be symbolic or not. The declaration of an ordinary identifier produces no declaration in the generated program.

One of the most powerful generation mechanisms is the procedure mechanism applied to non-symbolic procedures. When a non-symbolic procedure is called, the copy rule is applied, and the resulting statement is processed as if it occurred at the same place in the program being executed. This gives rise to various effects, e.g., input of external data, changing the values of ordinary variables, output of data), and to the production of a statement, if the statement resulting from the copy rule contained symbolic identifiers. If he procedure is recursive, the statement produced can have an infinity of forms, depending on the point from where it is called, and on external data. Conversely, the use of recursive procedures allows the production of any statement whose structure depends on external data, provided the rule of dependence is a computable function.

In the following formal definition, the processing of declarations, statements and expressions is completely defined. Besides other effects, a value is defined for each of them (which defines the value of a program as a special case of the value of a statement).

Furthermore, a generalized type is defined for each expression. Since the value of an expression is, in general, an expression containing identifiers, the types of ALGOL must be generalized. Just as, in ALGOL, the type *integer* is distinguished from the type *real*, the types *integer, real, Boolean* are distinguished, in macro-ALGOL, from the types *integer expression, real expression, Boolean expression*. In order to meet the efficiency requirement, the generalized types of expressions are defined in such a way that they can be determined at compile-time.

Declarations

Simple variables

An ordinary simple variable is declared and treated exactly like in ALGOL 60.[4] Its declaration has no value.

A symbolic simple variable is declared like a simple variable in ALGOL 60, except that its identifier is immediately preceded by one or more occurrences of the basic symbol *symbol*. The declaration of a symbolic simple variable has a value, which is the declaration obtained by deleting one occurance of *symbol*.

Examples:

integer x

declares x as an ordinary simple variable of type *integer*. This declarations has no value.

integer symbol y

declares y as a symbolic simple variable of type *integer*. The value of this declaration is:

integer y

The declaration

integer symbol symbol z

declares z to have the same properties as the above declared y. The value of this declaration is:

integer symbol z

Arrays

An ordinary array is declared and treated exactly like in ALGOL 60. Its declaration has no value.

A symbolic array is declared like in ALGOL 60, except that the array identifier is immediately preceded by one or more occurrences of *symbol*, and that any bound pair may be preceded by *macro*.

If the array identifier is preceded by more than one occurrence of *symbol,* the value of the declaration is the same declaration, where one occurrence of *symbol* has been deleted, and where all bound pairs are replaced by their values.

If the array identifier is preceded by exactly one occurrence of *symbol*, the value of the declaration is obtained by the following process:

1. The unique occurrence of *symbol* is deleted.
2. All expressions in bound pairs are replaced by their values.
3. If a bound pair is preceded by *macro*, the array identifier is replaced by a list of identifiers, one for each possible value of the corresponding subscript, and a note is taken of the correspondence. The bound pair with the symbol *macro* is deleted. The process is repeated until there remain no macro-bound pairs.
4. The result is the value of the initial declaration (with the reservation that, if no bound pair is left, the symbol *array* is deleted).

Examples:

real array A [1 : n]

declares A as an ordinary array identifier, and has no value (note that, if n is a symbolic identifier, the effect of this declaration is undefined).

real array symbol A [1 : n]

declares A as a symbolic array identifier. If n is a symbolic identifier, the value of the declaration is:

real array A [1 : n]

If n is an ordinary identifier whose current value is 3, the value is:

real array A [1 : 3]

Under the same hypothesis about n, the value of:

real array symbol A [*macro* 1 : n]

can be:

real A1, A2, A3

where the 3 identifiers are chosen distinct from any other declared identifier.

Procedures

If a procedure is declared with a type, the type may be *real, integer, Boolean, real expression, integer expression, Boolean expression.*

Besides this extension of type, an ordinary procedure is declared like in ALGOL 60. The declaration has no value.

A symbolic procedure is declared with one or more occurrences of *symbol* preceding the procedure identifier. The value of the declaration is a procedure declaration with the same procedure heading, and yhose body is the value of the body of the initial declaration. In the evaluation of the procedure body, all formal parameters are treated as symbolic identifiers.

Examples:

procedure P ; n : = n + 1

declares an ordinary procedure. (Note that n can be a symbolic identifier).

procedure symbol Q ; n : = n + a

If n is a symbolic identifier, and a an ordinary variable whose current value is 3, the value of this declaration is:

procedure Q ; n : = n + 3

Labels

A label declaration is the label identifier followed by a colon. The rules are the same as for simple variables.

Example: The value of :

symbol A : B : *symbol symbol* C :

is:

A : *symbol* C :

preceding the value of the statement following the initial declaration.

Switches and own variables

Switches and own variables are not considered here.

Expressions

We assume that a designational expression always reduces to a label.

The type of an expression other than designational can be:

integer	: its value is an integral number;
real	: its value is a real number;
Boolean	: its value is a logical value;
integer expression, real expression, Boolean expression	: its value is an expression of the indicated type in which all identifiers are symbolic.

The type and value of an expression are defined recursively as follows:

1. The type and value of a number or logical value are as defined in ALGOL 60.

2. The type and value of an ordinary simple variable are as defined in ALGOL 60.

The value of a symbolic simple variable is the variable itself. Its type is:

Boolean expression if the declared type of the variable is *Boolean;*

integer expression if the declared type of the variable is *integer;*

real expression if the declared type of the variable is *real.*

3. The type and value of a subscripted variable with an ordinary array identifier are as defined in ALGOL 60.

The value of a subscripted variable with a symbolic array identifier is a subscripted or simple variable obtained by evaluating all subscript expressions, removing the macro-subscripts if any, and replacing the array identifier by the identifier corresponding to the values of the macro-subscripts. It is defined only when all expressions in macro-subscript positions are of type *real* or *integer.* Its type is determined the same way as the type of a simple variable.

4. The type of a function designator with an ordinary procedure identifier is the declared type of the procedure. Its value is the value of the expression obtained by application of the copy rule.

If the procedure identifier is symbolic, the value is a function designator with the same procedure identifier, and the values of the actual parameters as actual parameters. The type is:

Boolean expression if the type of the procedure is *Boolean* or *Boolean expression* etc. . .

5. The value and type of an expression consisting of a unary operator applied to an expression are defined as follows:

- if the operand expression is of type *Boolean, integer* or *real,* the rules of ALGOL 60 are applied;
- if the operand expression is of type *Boolean expression, integer expression* or *real expression,* the type of the expression is the type of the operand. The value is equivalent to the expression consisting of the same operator applied to the value of the operand.

6. The value and type of an expression consisting of a binary operator applied to two expressions are defined as follows:

- if both expressions are of type *Boolean, integer* or *real,* the rules of ALGOL are applied;
- if at least one of the two expressions is of type *Boolean expression, integer expression* or *real expression,* the value is an expression equivalent to the expression consisting of the same operator applied to the values of the operands. The type is always *Boolean expression, integer expression* or *real expression,* the first symbol being determined by the rules of ALGOL 60.

7. The value and type of a conditional expression are determined as follows: let

if B *then* E1 *else* E2

be the conditional expression;

- if B is of type *Boolean,* the value is the value of E1 if the value of B is *true,* the value of E2 if the value of B is *false.* If E1 is a Boolean expression the type is the type of E1 VE2. If E1 is an arithmetic expression, it is the type of E1 + E2.
- if B is of type *Boolean expression,* the value of the expression is:

if VB *then* VE1 *else* VE2

where VB, VE1, VE2 are the values of B,E1, E2. The type is determined as in the first case.

8. The value of an ordinary label is undefined. The value of a symbolic label is the label itself.

9. *Remark:* The value of an expression is defined up to an equivalence. It is therefore left to each implementor to decide whether, for example:

+A is to be replaced by A;
−(−A) is to be replaced by A;
A + 0 is to be replaced by A;
2 + 3 is to be replaced by 5 (where 2 and 3 are the values of expressions of type *integer expression*), etc. . .

Statements

Since a statement has always a value, we shall speak indifferently of execution or evaluation of a statement.

1. Assignment statement

If all left part variables are ordinary variables, and if the right part expression is of type *integer, real* or *Boolean,* the statement is executed according to the rules of ALGOL 60. The value is a dummy statement.

If all left part variables are symbolic, the value is an assignment statement with the values of left part variables as left part variables, and the value of the expression as right part expression.

In all other cases, the effects and value of an assignment statement are undefined.

2. Go to statement

If the label is an ordinary label, the statement is executed as in ALGOL 60. Its value is a dummy statement.

If the label is symbolic, the value is a goto statement with the same label.

3. Conditional statement

If the Boolean expression is of type *Boolean,* the first statement or second statement (if any) is executed.

If the Boolean expression is of type *Boolean expression,* the components of the conditional statement are evaluated in sequence. The value is a statement equivalent to the conditional statement built with these components.

4. Procedure statement

The rules are the same as for a function designator, except that no type is defined.

5. For statement

Let

for V : = e_1, e_2, . . . e_n *do* S

be the for statement.

If V is a symbolic variable, all expressions in the for list and the statement S are evaluated. The value is a for statement with the values of these components as corresponding components.

If V is an ordinary variable, and if n > 1, the value is that of the statement:

begin for V : = e_1 *do* S ; . . . ; *for* : = e_n *do* S *end*

If n = 1, 3 cases are possible:

(1) e_1 is an arithmetic expression. The value is the value of the statement:

begin V : = e_1 ; S *end*

(2) e_1 is of the form A *step* B *until* C. The value is the value of the statement:

begin V : = A;
L1 : *if* (V-C)x sign (B)>0 *then goto*
L: Element exhausted;
S ; V : = V + B; *goto* L1;
Element exhausted:
end

(3) e_1 is of the form E *while* F. The value is the value of the statement:

begin L3 : V : = E; *if* ¬ F *then goto*
Element exhausted;
S; *goto* L3;
Element exhausted:
end

6. Dummy statement

The value is a dummy statement.

7. Compound statement

The statements of the sequence are evaluated from left to right, until a go to statement with an ordinary label is encountered, or the sequence is exhausted.

In the first case, if the label is declared inside the compound statement, the evaluations are resumed from the point where it is declared. If the label is declared outside, or in the second case, the evaluation of the compound statement is completed, and its value is equivalent to a compound statement made of the sequence of values obtained thus far.

8. Block

The declarations are evaluated. If there is at least one value declaration, the value is a block with the value declarations as declarations and the value of the statement sequence, treated like a compound statement, as statement sequence. If there is no value declaration, the value is equivalent to the value of the statement sequence treated like a compound statement.

Examples

1. Generation of a polynomial. We assume that the degree and coefficients of the polynomial are unknown when the generating program is written. The degree is the current value of the ordinary variable n, the coefficients are the elements of the value of the ordinary array a [0 : n], and the variable is the symbolic identifier x.

We write

p (x, a, n)

where we want the polynomial to be generated: the polynomial is the value of the above function designator. The procedure p has been declared thus:

real expression procedure p (X, A, N);
value N; *real expression* X; *array* A; *integer* N;
p : = *if* N = 0 *then* A [0] *else* p (X, A, N-1)× X + A[N]

If the current degree is 2, and if the current coefficients are 1, 2, 3, the application of the copy rule to p (x, a, n) will yield

if n = 0 *then* a [0] *else* p (x, a, n-1)× x + a [n]

ince 'n = 0' is of type *Boolean,* and its value is *false,* the value of this expression is the value of:

p (x, a, n-1)× x + 1

(since 1 is the value of a [2]).

The value will be:

(3 × x + 2) × x + 1.

2. Generation of the statement:

if B (1) *then* S (1) *else*
if B (2) *then* S (2) *else*
— — — — — — — —
if B (n) *then* S (n) *else* error

where n is unknown. We assume that B (n) and S (n) are computable and defined recursively. This statement is the value of the procedure statement:

q (1, n)

where q is declared thus:

procedure q(k,n); *value* k,n;
integer k, n;
if k = n + 1 *then* error *else if* B (k)
then S (k) *else* q (k + 1, n)

Comments

1. Implementation

The compiler of macro-ALGOL will produce machine-code for all executable parts of the macro-program, and calls to a subroutine for operations of symbolic arithmetic and program generation.

It is important to note that all symbolic results are always on top of the stack, since there are no variables with symbolic values.

The execution of the machine-code produced by the compiler never calls for a compilation process: this

is because the symbolic nature of an identifier never changes during execution of a program. This would not be the case with a language which would have a self-generation facility (i.e. the possibility, for a program, of generating itself during its execution).

2. Completeness

A macro-language is complete if any possible program generation algorithm can be programmed in this language.

As defined above, macro-ALGOL is certainly not complete. Its main weakness is in the generation of declarations, where the only non-trivial mechanism is that of macro-bound pairs, which provides only for the generation of an unknown number of simple variables, or arrays of one same dimensionality and size.

Part of the necessary extensions could be used in direct programming: these extensions would be general mechanisms of recursive definition of declarations, which would yield a means of declaration of arbitrary composite types.

From another point of view, the fact that variables of types *integer expression,* etc. . . , do not exist, might seem to make the language essentially incomplete. However, the introduction of such variables is probably not necessary. The reason for this is that a variable is fundamentally necessary only when a result must be used more than once. A symbolic result must appear, sooner or later, in the generated program, and it is not good practice, in general, to write a given expression or statement more than once in a program.

It remains that, in the first example above, the polynominal could be more efficiently generated by a for statement than by a recursive procedure. However, the auxiliary variable which would take on, as values, polynomials of increasing degrees, would not serve the general purpose of a variable: its value would always be used only once. Some kind of 'for expression,' leaving the iterated result anonymous, would be a better solution, and could be used to advantage in direct programming as well.

REFERENCES

1 T. E. CHEATHAM, JR.
The introduction of definitional facilities into higher level programming languages
AFIPS, 29, FJCC (1966)

2 B. M. LEAVENWORTH
Syntax macros and extended translation
Comm. A.C.M., 9, 11 (1966)

3 P. J. LANDIN
The mechanical evaluation of expressions
Computer Journal 6, 4 Jan. (1964)

4 J. W. BACKUS, et al.
Revised report on the algorithmic language ALGOL 60
Ed. P. Naur Comm. A.C.M. 6,1 (1963)

COMMEN: A new approach to programming languages

by LEO J. COHEN

Leo J. Cohen Associates
Trenton, New Jersey

INTRODUCTION

The *C*ompiler *O*riented for *M*ultiprogramming and *M*ulti-processing *EN*vironments is an outgrowth of the FADAC Automatic Test Analysis Language (FATAL) developed for the University of Pennsylvania under their contract with the Frankford Arsenal to investigate techniques for the automatic check-out of electronic circuits. The purpose of the FATAL Compiler was to prepare programs for execution in the FADAC Computer. This computer has a two-wire teletype input-output system that can be used to establish connections between a unit under test and a set of test equipments through an interface device. The configuration for the check-out system is shown in Figure 1.

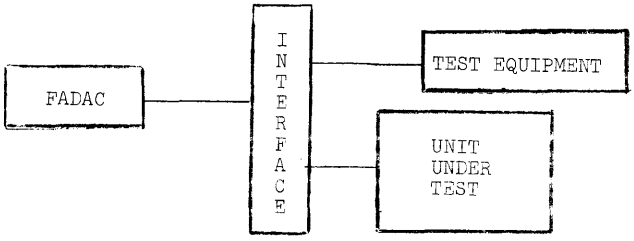


Figure 1 – FATAL checkout configuration

Numerical and logical techniques for the automatic check-out of electronic circuits were extensively developed.[1] Their application to automatic check-out by the FADAC Computer required the development of a programming language[2] that would allow the expression of both numerical and logical techniques necessary. In addition, the programming language necessarily had to provide statements for accomplishing the various connections and disconnections between the test equipment and the unit under test. The difficulty in the development of the FATAL compiler derived from the fact that the interface device was not in existence and the commands for effecting the various check-out functions were unknown. As a consequence it was decided to develop the FATAL compiler along lines that would permit new statements to be specified as desired. A further objective was to allow the specification of the meaning of these new statements via the already existing FATAL language.

The COMMEN compiler is an extension of the techniques developed in the FATAL project. The name for this compiler derives from certain invaluable properties that it possesses relative to multi-access systems (multi-programming, time sharing) and from its unique capability for automatically developing the control of a parallel process. The arrangement of the name into the acronym "COMMEN" was designed to indicate its most important application as a language unification technique, providing a single language for system command, computer programming, program documentation, and user-program dialogue.

The syntactical categories

The philosophical approach to the COMMEN compiler is perhaps unique in that it implements the translation of four mutually exclusive syntactical statement types. This is in opposition to compilers which determine a set of statements for which an implementation is provided.

The four syntactical categories for the statement types are declarators, assignments, procedure calls and verbs. Statements falling into the second and third syntactical categories are familiar from ALGOL, and generally adhere to the structural rules and techniques of application defined there. The syntactical form for declarators in COMMEN is an alphanumeric string terminated by ",". The alphanumeric string-plus-comma is interpreted as the declarator operator, and the alphanumeric string as the name of the declarator. Thus, the names of the declarators of the

system are not reserved words. In fact, there are no reserved words at all in the system so that the user has complete freedom in the construction of new statements.

$$A_1(P_1)A_2(P_2)....A_n(P_n)....$$

A_i = i^{th} alphanumeric string

P_j = j^{th} parameter set

Examples

```
IF (A) GTR (B)
FOR (CNT) EQL (Ø,5,5*(A+B) )
COMPARE (VALUE) TO (AX-B,AX+B)
COMPARE (VALUE) TO (AX) + OR - (B)
```

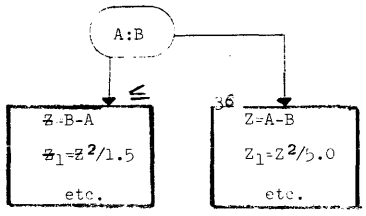
Figure 2 – General verb syntax

The most important syntactical category is for the set of statements that we have defined as verbs. In particular, verbs differ from declarators in the following sense. Declarators may be considered as commands from the user to the compiler for compiler time action, whereas verbs may be considered as commands from the user to the computer itself for object time action.

The syntactical form of a verb is described as an indefinite sequence of character strings and parameter sets enclosed in parentheses. A parameter set is defined as a string of parameters separated by commas. Figure 2 gives a pictorial representation of the general syntax for verbs and several examples. Note that though these examples of verbs may be suggestive of the functions that they represent, their exact meaning has yet to be specified. Since the functions that a compiler language statement offers to the user are defined in terms of the object machine code that implements it at execution time, the technique for the specification of a new statement in COMMEN is to allow the user to provide the necessary programming for his new statements. Further, it allows him to provide this programming in the form of COMMEN statements. In the subsequent sections of this paper this idea is made more specific, and implications for programming are developed in an example.

Phrase structure

The notion of phrasing and phrase structure is of basic importance in a compiler language if it is to allow the user a high degree of similarity between his flow chart and his compiler language program. In general, a phrase is defined as a single statement, or as a sequence of statements that are designated collectively as a phrase.




```
Phrase structure                          Phrase Structure
   Fortran IV                                 Algol

     IF (A .GT. B) GO TO 36                IF A GTR B;
     Z=B-A                                 BEGIN;
     Z1=Z**2/1.5                               Z=A-B;
                                               Z1=Z**2/5.0;
          :                                    GO NEXT:

     GO TO 37                              END;
30   Z=A-B                                 Z=B-A;
     Z1=Z**2/5.0                           Z1=Z**2/1.5;
          :

37   (program continues)                   NEXT.. (program continues)
```

Figure 3 – Phrase structure, ALGOL & FORTRAN

```
IF (A) GTR (B)
BEGIN,
      Z=A-B
      Z1=Z**2/5.0
      GO (NEXT)
END,
Z=B-A
Z1=Z**2/1.5
   .
   .
   .
NEXT..          (program continues)
```

Figure 4 – Phrase structure, COMMEN

The crudest form of phrase structure is found in FORTRAN IV where only the statement following an IF is treated as a phrase. A second example from FORTRAN is where the collection of statements within a DO loop is treated as a phrase.

An example of phrase structure in FORTRAN IV is shown in Figure 3 along with a comparative example of phrase structure as it is employed in ALGOL. In Figure 4 the phrase structure technique used in COMMEN for the same example is shown. As can be seen, the technique for phrase structure in COMMEN is virtually identical to that used in ALGOL with the two examples differing essentially in the syntactical structure of the verbs (IF and GO). This is to show that in general the two languages

agree on the application of phrasing relative to the control statements that use them. Thus, in these two examples for ALGOL and COMMEN, the IF implements an implied reference to the first, as well as to the second, phrase following this statement.

The notion of first, second, etc., phrase following a statement is of basic importance in the development of the COMMEN language technique. This importance derives from the fact that in order to specify the relationship between a new statement that the user is constructing and the phrases to which he might wish it to relate, we must have some concept of relative phrase location, or as it will be used in what follows, implicit phrase address.

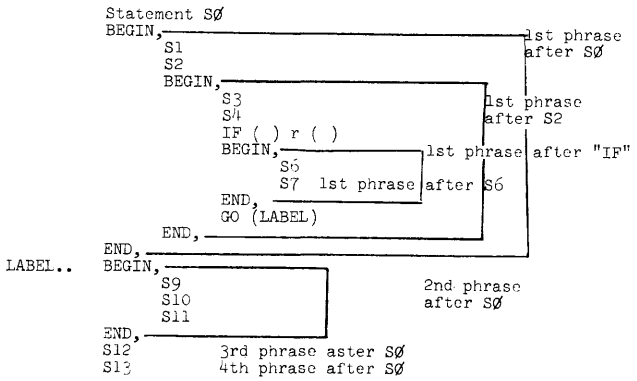


Figure 5 – Implicit phrase addresses

Figure 5 is an example of a collection of statements that have been organized into phrases and subphrases and which consequently determine relative phrase relationships. In this example statement Sϕ is followed by four phrases, so that the third phrase following statement Sϕ is actually the statement S12. Within the first phrase following Sϕ there are three phrases consisting of the statements S1 and S2, and the 10 statements concatenated into the second phrase following S1. Thus, by virtue of position, each phrase at a certain level of the nesting may be said to be in a position relative to any particular statement at that same level. This position relative to a statement will be referred to as the implicit phrase address of this phrase for that statement.

The other type of addressing possible in a compiler language program is the explicit phrase reference wherein a label (an alphanumeric string terminated by "..") is used to designate a phrase. References to such a phrase may be made via an explicit reference to the alphanumeric string associated with the label, as in the GO verb of the present example.

Some examples of COMMEN verbs which are familiar from ALGOL and which contain references to the first and second phrases following their application are shown in Figure 6. In the first of these examples, the FOR verb refers to the first phrase following its use as the one over which the control of iteration is exercised. Once the iteration has been completed the implementation of the verb causes transfer to the second phrase following the statement. The first and second phrases following the UNTIL and IF verbs are utilized in a similar way, and as in the case of the FOR, their exercise is designated implicitly as part of the meaning of the verb itself. It should be pointed out here, of course, that within the first phrase following any one of these three verbs, an explicit transfer may be to some other phrase of the program.

```
FOR (integer identifier) EQL (initial value , step, final value)

UNTIL (expression) relation (expression)

IF (expression) relation (expression)

        relations
           GTR
           GEQ
           EQL
           NEQ
           LEQ
           LSS
```

Figure 6 – Some COMMEN verbs

Specification of new verbs

The language of ALGOL is open ended to the extent that the user is allowed to declare and define new procedures in his program. A declarator is used to notify the compiler of the name of the procedure and of the formal parameters that are used when it is called. The declaration of the name for the new procedure is then followed by compiler language programming which supplies a definition of the procedure, and that in turn is followed by a termination declarator which informs the compiler that all definition programming for the procedure has been received. Once a procedure has been completely declared and defined to the compiler, the user is free to call this procedure in procedure call statements. Thus, this technique now allows the user to construct new statements of a specified syntactical form known as the procedure call.

The specification of new verbs in COMMEN follows much along the lines of procedure specification. In this case the language of the verb, along with the formal parameters for the verb, are declared to the compiler via the VERB, declarator. This declaration is the notification to the compiler that the programming that follows is the definition of the verb, and that this definition will be completed by a termination declarator. The statements used for defining the meaning of a new verb may be any of the statements

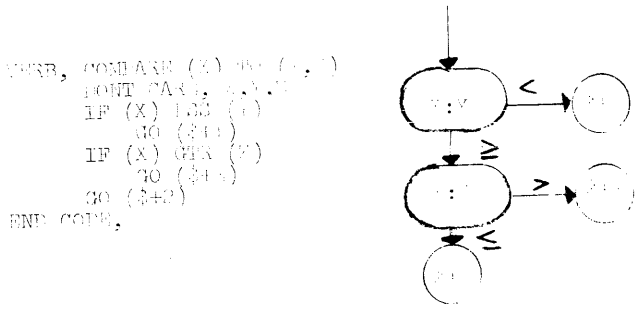


Figure 7 – Declaring a new verb

in the four syntactical categories, but of course may not include verb statements which have yet to be defined by the user to the compiler.

An example of the declaration of a new verb is shown in Figure 7. The language of the verb which we wish to declare is COMPARE (X) TO (Y,Z) where X, Y and Z are formal parameters that may be replaced on use of the verb by actual parameters, as in the case of the actual parameters for procedure call statements. The statement form is declared to the compiler via the declarator VERB, which also signals that all ensuing statements up to the occurrence of the declarator END CODE, serve to define the meaning of this new verb. For this example, the meaning of the verb is given by the logic of the flow chart shown in the figure. The connectors in this flow chart refer to the first, second and third phrases following the use of the new verb. As such they represent symbolic references to these three phrases and will become implicit phrase addresses in the definition of the verb. Thus, the logic for the new verb requires that the first, second or third phrase following the occurrence of the new verb is executed if X lies respectively to the left of, within, or to the right of, the closed interval (Y,Z).

In this example, the first statement following the declaration is the DONT CARE, declarator. It declares the formal parameters X, Y, Z to be either of floating or integer type. The next five statements encode the logic of the flow chart and refer to the symbolic phrase references as indicated. When the compiler recognizes the VERB declarator it stores away the format and language of the new verb in a verb library location. It then continues the compilation of the verb definition statements that follow, but diverts the result of the compilation of the statements to an area associated with the new verb. Thus the compiled version of the verb (plus information relative to its formal parameters and the symbolic phrase references) appears in the verb library and is available for application where required in the program.

```
          COMPARE (VALUE) TO (LOWER, UPPER)
($+1)     BEGIN,
                -
                -
                -
          END,
($+2)     BEGIN,
                -
                -
          END,
($+3)     BEGIN,
                -
                -
                -
                -
          END,
```

Figure 8 – Object time use of COMPARE

With the new verb declared, the programmer is free to use it as he requires for the implementation of his flow chart. Figure 8 shows a pictorial example of the use of this new verb with actual parameters, and the three following phrases to which it implicitly refers. When this verb call is translated, the compiled programming for the verb that was placed in the verb library is now inserted in-line in the output code. The actual parameters are substituted for the formal parameters and the symbolic phrase addresses are translated into the equivalent of assembly language references to symbolic tags that will later be associated with the phrases to which they refer.

Compounding verbs

Thus far the verb appears as the analog of the macro instruction in an assembly language with the major difference that the verb may implicitly refer to other phrases in the program that have a position relative to the verb. The real power of this technique at the compiler language level however, lies in our ability to compound the language of verbs into a very high level, problem oriented language that may serve directly as the formal documentation for the program for the non-programming technician.

In general the technique is as follows: Starting from a basic set of verbs, they are used to define new verbs which expand the set. These in turn are used for still newer verbs, until the language of the resulting verbs is in suitable agreement with the language of the problem. In order to provide a sufficient degree of generality, COMMEN accepts an OWN CODE, declarator which allows the insertion of assembly language programming. This declarator may appear within verb definition code.

The technique described above is illustrated by the following example. Suppose we wish to make available a statement by means of which we may determine whether or not a given variable is within a specified range of a second variable. Let us choose

the following language for our verb, using the dummy parameters X1, X2, and X3; COMPARE (X1) TO (X2) + OR − (X3). Explicitly, we want this verb to transfer control to the first phrase following its occurrence if X1 is less than (X2 − X3), to the second phrase following if X1 is within the closed interval (X2 − X3, X2 + X3) and to the third phrase following if X1 is greater than (X2 + X3). Figure 9 shows the necessary programming for the declaration and definition of this new verb.

```
VERB, COMPARE (X1) TO (X2) + OR-(X3)
DONT CARE, X1, X2, X3
      COMPARE (X1) TO (X2 - X3, X2 + X3)
      GO ($+1)
      GO ($+2)
      GO ($+3)
END CODE,
```

Figure 9 – Compounding verbs

```
VERB, SKIP NEXT PHRASE IF (X) ...
      WITHIN 10% RANGE OF (Y)
      REAL, X, Y,
      COMPARE (X) TO (Y) + OR - (Y/10.0)
      GO ($+1)
      GO ($+2)
END CODE,
```

Figure 10 – The SKIP verb

Notice that the new verb used the previously declared COMPARE with properly selected parameters. The example assumes that this verb already exists in the verb library. Since the definition for a verb is in essence a self contained program whose only relationship to the environment in which it appears is through its formal parameters and its symbolic phrase references, the three transfer statements will be necessary to complete the definition of the new verb. This new form of the COMPARE verb is used to construct the compound shown in Figure 10. In this example the elision mark "..." is used to signify the continuation of a statement onto the next line.

```
VERB, EXECUTE (LABEL) IF (X) ...
      OUT OF 1000 RANGE
      REAL, X
      LABEL, LABEL
      COMPARE (X) TO (900.0, 1100.0)
      GO (LABEL)
      GO ($+1)
      GO (LABEL)
END CODE,
```

Figure 11 – The EXECUTE verb

As an example of the high degree of problem orientation that can be achieved by this process consider the verb defined in Figure 11. For this verb the first formal parameter, LABEL, will be the name, i.e., explicit address, of some phrase in the program. The second formal parameter for the verb is the variable X, and for this case is defined to be a real (floating decimal) number. Our new COMPARE verb is then applied to the variable X along with the constants shown. If the value of X is out of the 1000 range, i.e., lies either to the left or to the right of the range, then a transfer of control is made to the selected phrase. Otherwise, if the value of X lies within the 1000 range a transfer is made to the next phrase following the occurrence of this verb.

```
TEST 1.. EXECUTE (ERROR 1) IF...
         (341.5X+K1) OUT OF 1000 RANGE

TEST 2.. EXECUTE (ERROR 2) IF...
         (395.8X-K1) OUT OF 1000 RANGE

         etc.
```

Figure 12 – Example of language orientation

The application of this verb in a general testing environment might be as shown in the example Figure 12. Here the just defined verb is used with various labels for the first actual parameter, each representing a specified routine to be executed in the event that the value of the second actual parameter is out of the 1000 range.

CONCLUSION

It was originally felt that the real value of compilers derived from the amount of labor that they saved the programmer by cutting back the number of instructions that he had to write in the compiler language relative to the number of instructions necessary to do the same job written in assembly language. As compilers came more and more into popular acceptance it has been readily recognized that their true value lies in the verbal facility they provide for the expression of a problem in a form that may then be translated into an operating computer program. More particularly, it is often the method used to express our understanding of a problem that shapes and informs our method of approach to its solution. Therefore, the higher the degree of problem orientation in the language with which we represent a problem for translation into computer executable terms, the closer we come to intellectual symbiosis with computing machines. The various simulation languages available are excellent examples of this.

The development of COMMEN is another step in this direction. As a programming technique its objective is to reduce to a minimum the difficulties inherent in expanding the problem expression language. At its ultimate application, it allows us to unify the several languages required to program, document, command, and converse with our large scale com-

puter systems. As an example of the application of COMMEN to the language unification problem, see reference 3.

REFERENCES

1 Study of piece part fault isolation by computer logic V.3 Circuit analysis solvability and systems studies ICR Univ of Pa Philadelphia Pennsylvania 1964

2 Study of piece part fault isolation by computer logic V.1 User's Manual-FATA compiler assembler system ICR Univ of Pa Philadelphia Pennsylvania 1964

3 Additions to COMMEN for Time Sharing Time Sharing Workbook Inst for Advanced Tech C-E-I-R 1200 Jefferson Davis Hwy Alexandria Virginia

SPRINT a direct approach to list processing languages

by CHARLES A. KAPPS
University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

Most current list processing languages such as LISP and IPL-V operate in an indirect manner, i.e., during execution of a program written in these languages, the basic operations do not deal directly with data but rather with addresses which point to the data, making it awkward to perform operations at the input syntax level.

Other languages, such as L6, give the programmer a closer relationship with data on which his program operates, but are highly machine oriented languages, and therefore are really effective on only a small number of machines.

The aims of SPRINT are to give the programmer direct access to both data and program in a way that is as nearly machine independent as is possible. In particular, SPRINT is completely unstratified. It has divorced itself from any machine oriented addressing schemes, number schemes, and word formats. It allows the programmer direct access to a data scheme which is as general as possible from the human point of view, while keeping in mind some degree of practicability.

In order to achieve these goals some sacrifice had to be made with respect to machine efficiency. SPRINT is intended to reflect the thought processes of the programmer, not the logic of some machine currently in existence which will be obsolete in a few years. In a way SPRINT was modeled as the machine language for a futuristic machine, one where the hardware was designed to employ a more humanly, logical structure than those of today. The result is that SPRINT would operate much more efficiently if mechanized on machines which have more advanced organizations than those of today.

Basic properties of SPRINT

The "word," being the basic unit of information, consists of a variable length string of alphanumeric characters. (In order that some semblance of efficiency be maintained, there is an upper bound on the length of a word. In the case of present mechanization, this bound is 179 characters, which seems adequately long for most purposes.) SPRINT words have no meaning in themselves, and take on meaning only when interpreted by the programs which use them. This is an important feature of SPRINT, since it leads to non-stratification of the language.

Arithmetic information in SPRINT is denoted by any word composed solely of numeric characters, and represents the corresponding decimal integer.

For purposes of operating the execution cycle, the programmer must mark each SPRINT word with a class mark indicating whether it is to be treated as an instruction or as data when it is encountered by the instruction decoder. This is analogous to the use of the operator "QUOTE" in LISP.

Several SPRINT words (about 35) have been set aside to name "basic instructions." Basic instructions are operations which have been programmed in machine language, or which would be in the hardware of an actual machine. Figure 1 gives a list of some of the more important basic instructions of SPRINT.

Linearly ordered sets of SPRINT words form "lists." Lists may contain either programs or data, but since SPRINT is unstratified, program lists and data lists are indistinguishable except in context. Lists have names, which consist of a single SPRINT word, and SPRINT programs refer to the lists by means of their alphabetic names. Lists are stored in an associative type of memory area, and are located when sought, by means of their names. It is important that these names are preserved, since this allows SPRINT programs to call for lists which are named

MNEMONIC	MEANING	ARGUMENTS REMOVED FROM THE OPERAND STACK	RESULTS RETURNED TO THE OPERAND STACK	OTHER ACTIONS
RDT	Read Data	W	----------	An input card is read and stored as a list named "W".
WDT	Write Data	W	----------	The list named "W" is printed out.
ADD	Add	A, B	A + B	----------
SUB	Subtract	A, B	B - A	----------
MPY	Multiply	A, B	A x B	----------
DIV	Divide	A, B	B mod A and B/A	----------
REV	Reverse	W, V	V and W	----------
REP	Repeat	W	W and W	----------
DEL	Delete	W	----------	----------
CON	Concatenate	W, V	Concatenation of W and V	----------
DCN	Deconcatenate	A, W	First A characters of W and the rest of W.	----------
BNG	Bring	W	The entire list named "W"	
CLL	Call	W	----------	The entire list named by "W" is pushed into the instruction stack.
TRA	Transfer	W, V	----------	"V" is stored as a one-word list named "W".
STO	Store	A, W, V_1,..V_A	----------	"V_1" through "V_A" are stored as an A word list named W.
FND	Find	W	W	If W names a list in memory the next instruction is skipped, otherwise the next instruction is executed and the one following is skipped.
TZE	Text for Zero	W	W	If W belongs to 0^* the next instruction is skipped, otherwise the next instruction is executed and the one following is skipped.

A and B represent arbitrary numeric words; W and V represent arbitrary words.

Figure 1 – A partial enumeration of the basic instruction of SPRINT

by SPRINT words which are "computed" during execution. Basic instructions are available in SPRINT which allow any word of any list or any character of any word to be modified in any Turing computable fashion.

The execution of SPRINT programs takes place in an area consisting of two push down stacks, designated as the instruction stack and operand stack. Basic instructions in SPRINT do not have "addresses," but refer to the top several levels of the operand stack for their arguments and results. In addition, some instructions affect the instruction stack and the associative memory area. Figure 2 shows the basic instruction cycle of SPRINT.

It can be seen that Figure 2 is actually a flow chart for a suffix language processor; consequently, simple SPRINT programs will appear in suffix form.

For example, to evaluate the algebraic expression:

$$(((A+B)\,C+D)\,(E-10))$$

one could execute the following program list:

A B ADD C MPY D ADD
E ⊕10 SUB MPY

Here "⊕" indicates a data class mark, and instruction class marks are assumed elsewhere. "ADD," "SUB," and "MPY" are the SPRINT mnemonics for add, subtract, and multiply respectively, and "A," "B," "C," "D," and "E" are assumed to be either the names of data lists which contain a single numeric word in the data class, or SPRINT programs for computing a single numeric word.

Complex data structures within SPRINT

One of the main advantages of having an associative memory is that names can be created by com-

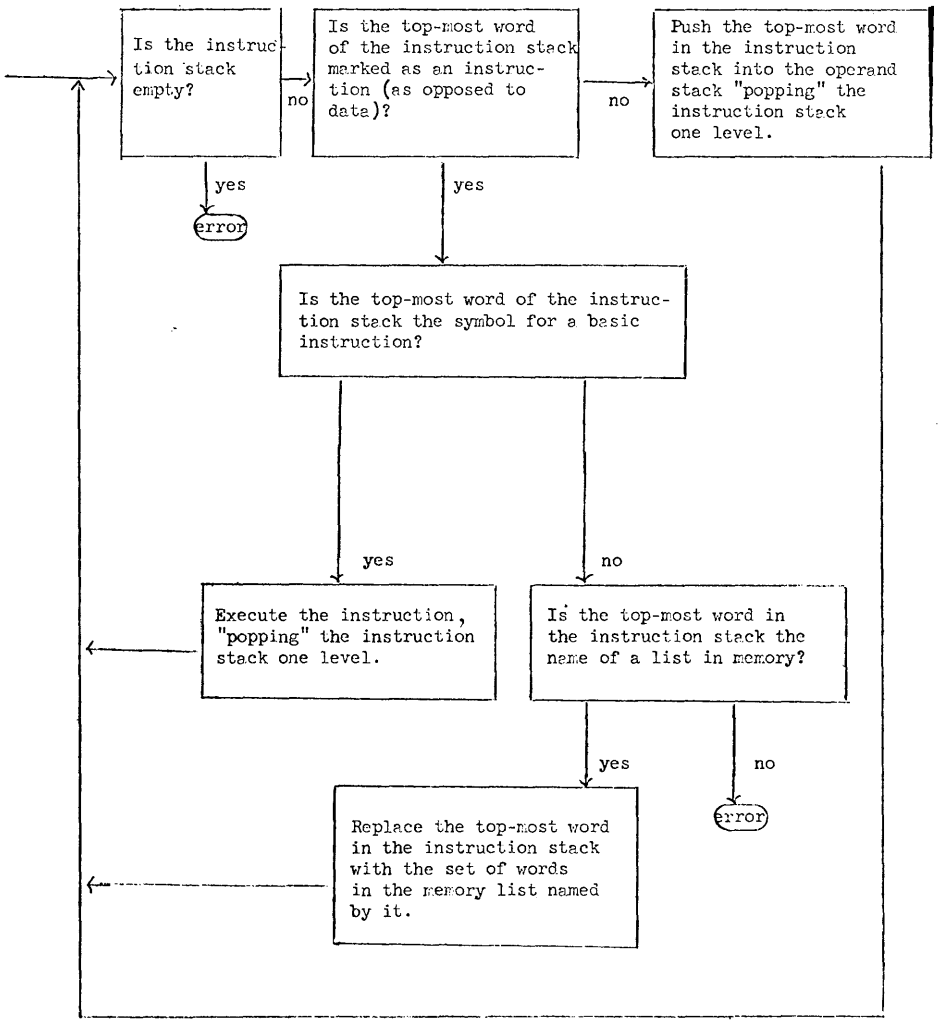


Figure 2 – Basic instruction cycle

putation during the execution program and can be used to store and retrieve information for that program. An example of how this is useful is the case of indexed arrays such as matrices.

The typical method for handling matrices in many programming languages is to use a symbol of the forms A(5, 3) to represent an element of a matrix. Since "A", "(", ",", ")", and numeric digits are all legal characters in the SPRINT alphabet, the concatenation A(5,3) is a legal SPRINT word, and thus should be used as the name for a data list of one numeric word representing the matrix element A(5, 3). As concatenation (symbolized by "CON") and decimal arithmetic are basic operations in SPRINT, the above scheme for storing the elements of a matrix can be programmed quite easily. For example, the SPRINT program list

⊕) J ⊕, I ⊕A(CON CON CON CON

would produce the "name" of the element A_{ij} for any numeric values of "I" and "J," i.e. if "I" and "J" were the names of lists containing the single numeric data words "5" and "3" respectively, the above program list would output the SPRINT word "A(5, 3)." (Note that the reverse order of concatenation is due to the suffix nature of SPRINT.)

It should be noted here that the above scheme for naming the elements of a matrix is completely independent of any knowledge concerning the size of the matrix. This gives us a completely dynamic storage allocation procedure for matrices which *never* requires the use of any array size declarations.

The above techniques employed with matrix notation can obviously be generalized to operate with arrays having any number of indices (the only limitation being the maximum length of a SPRINT word). A further generalization is to construct a data scheme where the number of indices itself is a variable. An example of this is canonical tree addressing.[1] Figure 3 shows an example of a simple tree with the addresses of each node given, which could be used in a SPRINT program exactly as shown, i.e., "A(2, 1, 2, 1)" would name a list containing the data to be associated with the lowest node on the tree in Figure 2.

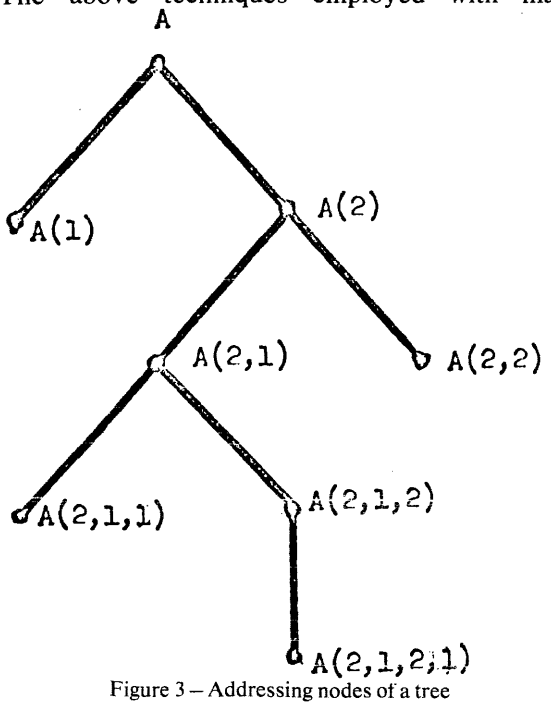


Figure 3 – Addressing nodes of a tree

Mechanization of the SPRINT language

The SPRINT language has been mechanized on two machines, the IBM 7094 and the IBM 7040. The 7040 version, being the latter, is somewhat improved over the 7094 version. Both operate in the interpretive mode and use a combination hash address linked list search for the associative memory. The push down stacks are mechanized by linear arrays of computer memory backed up by reels of magnetic tape or sequentially accessed disk memory, thus giving them practically unbounded length. Both versions of the interpreter contain roughly 3000 machine instructions and operate reasonably efficiently considering that a binary machine is not well suited for the mechanization of SPRINT. As an example, six minutes were required to compute the factorials up to 93 factorial (in full precision!). This required more than 4400 recursions and multiplications, many of which involved numbers with decimal precisions greater than 100 digits.

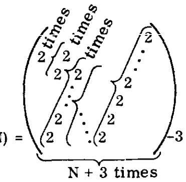
Although quite usable, the mechanization of the SPRINT interpreter is still in the experimental stage, and improvements and additions are continually being made.

A programming example

As an example of what SPRINT programming looks like and how it operates, the tradition for list processing languages was followed and a program for computing Ackermann's function was written. This program, however, differs from the classical Ackermann function routine in several aspects.

Firstly, Ackermann's function was defined not in its usual form[5] but rather in terms of an infinite class of *primitive* recursive functions, i.e.

Each of the expressions on the right is a primitive recursive function, and in fact consists of an operation which is a simple repetition of the previous operation. The first three of these operations can of course be programmed directly in SPRINT without "loops." The rest of course requires looping or recursive subroutines.

Function	
$A(0, N) = N + 1 = (N + 3) + 1 - 3$	successor
$A(1, N) = N + 2 = 2 + (N + 3) - 3$	repeated successor
$A(2, N) = 2 \cdot (N + 3) - 3$	repeated addition
$A(3, N) = 2^{(N + 3)} - 3$	repeated multiplication
$A(4, N) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{N+3 \text{ times}} - 3$	repeated exponentiation
A(5, N) = 	repeated repeated exponentiation, etc.

The result of programming Ackermann's function in this manner was a tremendous gain in efficiency. This program was able to compute values of Ackermann's function in a few seconds which upon calculation would appear to take millions of years using programs such as that in the IPL-V manual.[5]

At first glance, it would seem that a program of this sort, although efficient, could not be general since an infinite number of subroutines would be required to describe the infinite class of primitive recursive functions. Nevertheless, this program is completely general (ignoring such limitations as the fact that the SPRINT machine has to fit in the physical universe), since for any given computation only a finite number of the primitive recursive functions need be defined. This program is written to *write* as many subroutines as it needs to compute Ackermann's function for the arguments given.

This program then indicates many of the powerful features of SPRINT in that it is easily able to perform the following operations:

1. Test whether or not a given subroutine exists in its memory.
2. Perform computation for writing subroutines.
3. *Execute* subroutines after they have been written (or modified).
4. Produce an expanding hierarchy of information within its internal structure.

The program for computing Ackermann's function is shown in Figure 4. This is a listing of the input cards to the program; it is not directly in SPRINT but in a second level language. The input program which reads in these cards and compiles them into SPRINT is actually written in SPRINT, thus again illustrating the non-stratification of SPRINT. There are several meta-linguistic characters which have the following interpretation at "load time."

* introduces the name of a new list.

* introduces the name of a new list.

\+ and − introduce a word of data

/ introduces comments.

$ marks the end of input.

The absence of a special character indicates an instruction word.

Blanks separate words.

Figure 5 illustrates a sample of output from the program. Note that A(3, 75) would not only take a long time to compute using the IPL-V routine, but is also too large a number to be handled in most IPL-V mechanizations. Of course, the values of Ackermann's function become so large so quickly, that even the increased power of this routine gains little once the higher values of arguments are reached.

SUMMARY

SPRINT is a programming language which is being designed to be an experimental tool for studying problems in the theory of programming, and for testing algorithmic concepts. Since mechanizations of SPRINT on present machines must operate interpretively, in order that the language be completely unstratified, sacrifices are necessary with regard to the overall efficiency of the system. Much of the lack of efficiency could be overcome, however, if SPRINT were mechanized on a computer which had hardware facilities built in for performing some of the more complicated functions of SPRINT. Some work is currently being done in this area. In fact some machines presently have built in push down stacks, and decimal as well as binary arithmetic circuits, and studies are being made on hardware associative memories. Thus part of the work of the project for which SPRINT was developed was concerned with the design of a machine which would have SPRINT as its machine language.

Experience has shown that SPRINT is quite easy to learn and use. Persons of various backgrounds ranging from high school students to Ph.D. candidates have been able to use SPRINT with little difficulty. A recent example concerns a student who developed an operating syntax directed recognizer in about two weeks work, using SPRINT. This program was written "from scratch" without prior knowledge of SPRINT or the employed algorithms.

An example of a language which is semantically similar to SPRINT is the GIPSY system developed by W. H. Burge.[7] SPRINT, however, is a much more highly developed language, especially with respect to the use of SPRINT words to form complex data structures. This is due to the fact that SPRINT language is handled directly throughout execution.

ACKNOWLEDGMENTS

I would like to give thanks to Dr. Harry J. Gray, Dr. John W. Carr, III, Mr. Alvin Vivatson and Mr.

```
   069716   003  KAPPS, C. A.     SPRINT INTERPRETER OUTPUT

 READ IN ARGUMENTS
         +X        RDT
 COMPUTE ACKERMANN'S FUNCTION.
         +2        X         +3        ADD       REV       +0        CON
         DO        +3        SUB
 CREATE OUTPUT FORMAT, AND PRINT
         +)=       +X        BNG       +,        REV       +A(       CON
         CON       CON       CON       CON       +Y        TRA       +Y
         WDT       +         CLL
*DO      / ROUTINE FOR CALLING OR WRITING ROUTINE
         END       GEN       CLL
*GEN     / ROUTINE FOR STARTING THE PROGRAM WRITING
         REP       +NAME     TRA       MAKE
*MAKE    / ROUTINE FOR THE ACTUAL WRITING OF THE NECESSARY PROGRAMS
         +1        DCN       DEL       REP       +1        SUB       +GO
         BNG       CON       +NEXT     TRA       +GO       BNG       CON
         +THIS     TRA       +GOZ      BNG       +P        +THIS     BNG
         CON       +GTZ      BNG       +THIS     BNG       +3        STO
         +NEXT     BNG       REP       +THIS     BNG       +GHD      BNG
         +P        +THIS     BNG       CON       +8        STO       END
         MAKE      DONE
*DONE    DEL       NAME      CLL       / AFTER ROUTINES ARE WRITTEN, CALL THEM
/ DATA FOR MAKE
*GO      0         *GOZ      OZ        *GTZ      TZE
*GHD     +TEMP     TRA       REP       TEMP      +1        SUB
/ NON RECURSIVE OPERATORS
*00      REV       DEL       +1        ADD       *01       ADD
*02      MPY       *OZ       DEL       DEL       +1        $
```

Figure 4—SPRINT interpreter output

```
069716   003  KAPPS, C. A.     SPRINT INTERPRETER OUTPUT


A(0,0)=1
A(0,1)=2
A(0,2)=3
A(0,3)=4
A(1,0)=2
A(1,1)=3
A(1,2)=4
A(1,3)=5
A(2,0)=3
A(2,1)=5
A(2,2)=7
A(2,3)=9
A(3,0)=5
A(3,1)=13
A(3,2)=29
A(3,3)=61
A(4,0)=13
A(4,1)=65533
A(5,0)=65533
A(3,75)=302231454903657293676541
```

Figure 5 – Sample of output from the program

William Slemmer who contributed much to the development of SPRINT.

The work on SPRINT has been supported by the following:

Contract numbers

Rome Air Development Center, Research and Technology Center, Air Force Systems Command, Griffiss Air Force Base, New York, Contract No. AF 30(602)-2994, Project 5581, Task 558102.

and

U.S. Army Electronics Research Command, Fort Monmouth, New Jersey, Contract No. DA 88-043 AMC-02377(E).

REFERENCES

1 S GORN
Processors for Infinite Code of the Shann-Fano Type
Proceeding of the symposium on mathematical theory of automata
Polytechnic Institute of Brooklyn April 1962

2 H F GRAY C KAPPS ET AL
Interaction of computer language and machine design
Technical report No RADC TR 64 511 May 1965

3 C KAPPS
Basic programmer's manual for SPRINT 1
University of Pennsylvania
The Moore School of Electrical Engineering
Available upon request from the author December 1964

4 J MCCARTHY ET AL
LISP 1.5 programmers manual
MIT Press Cambridge Massachusetts 1962

5 A NEWELL Ed
Information proceeding language V manual
Prentice Hall
Englewood Cliffs New Jersey 1961

6 K C KNOWLTON
A programmers description of L6
Communications of the ACM
Vol 9 no 8 pp 661-625 August 1966

7 W H BURGE
Interpretation stacks and evaluation
Introduction to system programming
Edited by P Wegner pp 294-312 Academic Press London and New York 1964

Syntax-checking and parsing of context-free languages by pushdown-store automata

by VICTOR SCHNEIDER
Computer Science Center, University of Maryland
College Park, Maryland

INTRODUCTION

By way of two heuristic examples, this paper demonstrates an algorithm that constructs computer-realizable machines for syntax checking and parsing of computer programs constructed from context-free grammars. The algorithm followed constructs a pushdown-store, one-way automaton that corresponds to the context-free version of the grammar being considered.* This automaton has the property of accepting a string of symbols on its input tape if and only if that string belongs to the language of its grammar. In addition, it signals the source of error in an unacceptable string on its input tape as a result of the correspondence between states of the automaton and rules of the grammar.

If the automaton designed from some grammar is initially nondeterministic in operation, a further algorithm can be applied for constructing a deterministic version of the automaton. Those cases for which a deterministic version of the automaton cannot be constructed by this algorithm are conditions for the impossibility of checking the grammaticalness of its language in single, one-way scans. These cases further correspond to a sufficient condition for the language to be ambiguous.

Notation

In this paper, a pushdown-store automaton (PDS automaton) consists of;

1. A pushdown store, or tape which is written on from right to left and read from left to right, and an associated alphabet.
2. An input tape that is read from left to right in one scan, and an associated distinct alphabet.
3. A set of states, including an initial state S_0 and a nonempty set of final states.
4. A next-state relation M, to be defined.
5. A terminating criterion, to be defined.

The alphabet of symbols read on the input tape is represented by (possibly subscripted) lower-case letters, and the push-down-store alphabet is denoted by (possibly subscripted) upper–case letters.

Associated with the next-state relation M are the tape instructions (x,y), with

$$x,y \in \{O,1\}.$$

For

$(x,y)=(1,O)$,

the reading head of the input tape is moved over one space.

For

$$(x,y)=(O,1),$$

the leftmost symbol of the pushdown store is erased, and, either a new symbol is written in its place, or the read-write head moves one space to the right. The conditions that determine whether or not a symbol is written on the PDS tape will be discussed presently. Boundary symbols are assumed to exist on the input and pushdown-store tapes to prevent the respective tape heads from running off the ends of the tapes.

As an example of how the next-state relation will be defined, suppose that a_1 is a symbol of the input tape, and A_2, A_3 are symbols of the pushdown store. S_1 and S_2 are states of the automaton in this example. The relation

$$M(a_1 A_2, S_1) = S_2, A_3, (O,1)$$

means that, for this example, when the currently read input tape symbol is a_1, the currently read PDS tape symbol is A_2, and the automaton is in state S_1, the following occurs:

The automaton transfers into state S_2, the symbol A_2 is erased from the pushdown store, A_3 is written in place of A_2, and the reading head on the input

*Given a grammar having one or more context-sensitive rules, a context-free version can be constructed by striking off all the contexts from the rules. The new grammar generates a language containing the language generated by the old grammar.

tape continues to scan symbol a_1.
The relation

$$M(\varnothing\%,S_f)=S_f, \quad ,(0,0) \qquad (1)$$

means that no more symbols remain to be read on the input tape, the pushdown-store tape is blank, and the machine is in state S_f, one of a set of final states. The automaton remains in state S_f, the notation (O,O) indicating a halt in operation. Relation (1) is the termination criterion adopted in this paper. When this criterion is satisfied by a computation on some input tape, the tape is said to be accepted by the automaton, and its string is a valid string of the language.

As is shown elsewhere,* it is possible to reduce any context-free grammar to a weakly equivalent, normal-form grammar** whose rules are of the following three types:

$$A_i : := A_{i1}A_{i2} \qquad (2,a)$$
$$A_j : := A_{j1}a_{j2} \qquad (2.b)$$
$$A_k : := a_k \qquad (2.c)$$

In the rules of (2), capitol letters represent the nonterminal symbols of the grammar; i.e., those symbols A_v for which there is at least one rule of the forms (2,a), (2,b), or (2,c), with A_v on the left-hand side. The remaining symbols in lower case are terminal symbols; i.e., symbols that appear in some string of the grammar's language. Note that the alphabet of terminal symbols is disjoint from the alphabet of nonterminal symbols, because the condition defining nonterminals does not apply to terminal symbols.

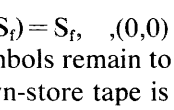
Algorithm for constructing the pushdown-store acceptor

By parsing the strings of a particular language, we mean the construction of tree diagrams over those strings.† The condition for well-formedness of a string of symbols with respect to a language then becomes the following one:

> If a string belongs to a language (is well-formed with respect to that language), a tree diagram over that string can be constructed.

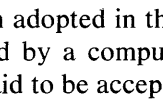
The existence of the tree diagram is a decidable property. We show the relation between constructing such a tree diagram over a string of some language and the computation of a pushdown-store automaton having that string on its input tape in what follows.

Consider the three forms of rules in (2) and how they are represented in tree diagrams. Rules of the form (2,c) are represented as shown in (3):



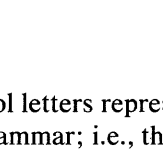
represents $A_k : := a_k$ (3)

Note that a_k is one symbol of the string being parsed, because it is a terminal symbol. Rules of the form (2,b) are represented as shown in (4):



represents $A_j : := A_{j1}A_{j2}$ (4)

Again, a_{j2} is one symbol of the string being parsed, and A_{j1} is some directly preceding nonterminal symbol of a partially constructed tree diagram representing the left-hand side of some rule applied earlier. Rules of the form (2,a) are represented as shown in (5):



represents $A_i : := A_{i1}A_{i2}$ (5)

As is shown elsewhere,[1] the nodes A_{i1} and A_{i2} of the partial tree diagram represented in (5) are adjacent in the sense that, before A_i links A_{i1} and A_{i2} in constructing the tree diagram, A_{i1} is the nearest node to the left of A_{i2} that is both unconnected to a node on its right and represents the left-hand side of a rule of the grammar. This property is naturally related to the convention of reading the topmost symbol of a pushdown store.

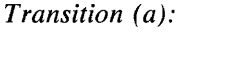
In constructing a pushdown-store automaton from the normal form of some grammar, three cases of transitions arise, corresponding to the three types of rules in normal form.†

Case I. Rules of the form (2,c), $U_j : := u_{j1}$.

Next-State Relation (a): $M(u_{j1}, S_o)=U_j, \quad ,(1,O)$

The transition corresponding to relation (a) above can be pictured as a transition in the state diagram of the PDS automaton:

Transition (a):



In (a), u_{j1} is read from the input tape. A transition occurs from initial state S_o to state U_j.

Next-State Relation (b): $M(u_{j1}, S_o)=S_o, U_j, (1,O)$

The transition corresponding to relation (b) can next be given in terms of the state diagram of the PDS auto-

*Reference 1, Chapter I.

**In many cases of interest, the normal-form grammar is strongly equivalent to the original grammar in the sense that the original parsing of a string of some language can be reconstructed directly from the parsing assigned to that string by the normal form grammar.

†*cf.* reference 2 for an introduction to this concept.

†Subcases (a) and (b) will be explained in what follows.

maton:

Transition (b):

$$S_o \qquad u_{j1}/U_j, (1,O)$$

In (b), u_{j1} is read from the input tape. Symbol U is written on top of the pushdown-store tape. A transition occurs to initial state S_0.

Transitions (a) and (b) of Case I correspond to adding a node and one branch to the partly constructed tree diagram over a string of some language, as shown in example (3) above.

Case II. Rules of the form (2,b), $U_j : := V_{j1}v_{j2}$.

Next-State Relation (a): $M(v_{j2}, V_{j1}) = U_j, \quad ,(1,O)$

The state-diagram transition corresponding to relation (a) is given by

Transition (a): $V_{j1} \qquad U_j$
$v_{j2}/(1,O)$

In (a), v_{j2} is read from the input tape. A transition occurs from state V_{j1} to state U_i.
Next-State Relation (b): $M(v_{j2}, V_{j1}) = S_o, U_j, (1,O)$

The state-diagram transition corresponding to relation (b) is given by

Transition (b): $V_{j1} \qquad S_o$
$v_{j2}/U_j, (1,O)$

In (b), v_{j2} is read from the input tape. Symbol U_j is written on top of the pushdown store (at the left end of the PDS tape string). A transition occurs from state V_{j1} to initial state S_o..

Transitions (a) and (b) of Case II correspond to adding a node and two branches to the partly constructed tree diagram over a string of some language, as shown in example (4) above.

Case III. Rules of the form (2,a), $U_j : := W_{j1}W_{j2}$.

Next-State Relation (a): $M(W_{ji}, W_{j2}) = U_j, (O,1)$

The state-diagram transition corresponding to relation (a) is given by

Transition (a): $W_{j2} \qquad U_j$
$W_{j1}/(O,1)$

In (a), symbol W_{j1} is read from the top of the pushdown store (leftmost symbol on the PDS tape) and erased. A transition occurs from state W_{j2} to state U_j.
Next-State Relation (b): $M(W_{j1}, W_{j2}) = S_o, U_j, (O,1)$
The state-diagram transition corresponding to relation (b) is given by

Transition (b): $W_{j2} \qquad S_o$
$W_{j1}/U_j, (O,1)$

In (b), symbol W_{j1} is read from the top of the pushdown store and erased. Symbol U_j is written in place of W_{j1} on the pushdown store. A transition occurs from state W_{j2} to initial state S_o. i_0.

Transitions (a) and (b) of Case III correspond to adding a node and two branches to the partly constructed tree diagram over a string of some language, as shown in example (5) above.

We now consider the circumstances under which subcases (a) and (b) are valid transitions of a parsing automaton. For cases I, II, and III above, if U_j appears on the right-hand side of a rule of the forms (6)

$$A_s : := U_j b_{s2} \quad \text{or} \quad A_t : := A_{t1}U_j \qquad (6)$$

transitions (a) appear in the constructed automaton. If U_j appears on the right-hand side of a rule of the form (7),

$$A_z : := U_j A_{z2} \qquad (7)$$

transitions (b) appear in the constructed automaton. In general, for the transition corresponding to a particular rule, both transition (a) and (b) may occur. The following examples will illustrate the construction of PDS automata directly from normal-form grammars, and will indicate how deterministic versions of nondeterministic PDS automata are constructed.

Examples in the construction of deterministic DPS automata

Example 1. A Grammar of Griffiths and Petrick.[3]

S : := AB B : := bc
A : := a B : := Bd
A : := ABb

The grammar of Example 1 is rewritten in normal form by introducing additional nonterminal symbols that are subscripted so that each nonterminal introduced is distinguishable from all other nonterminals in the grammar:

Grammar 1.1. Example 1 Rewritten in Normal Form

$S : := AB$ $\quad X_1 : := AB$ $\quad B : := X_2c$
$A : := a$ $\quad X_2 : := B$ $\quad B : := Bd$
$A : := X_1b$

By means of the correspondence between rules and automata states introduced earlier, Grammar 1.1 is translated into the PDS automaton of Figure 1.1.

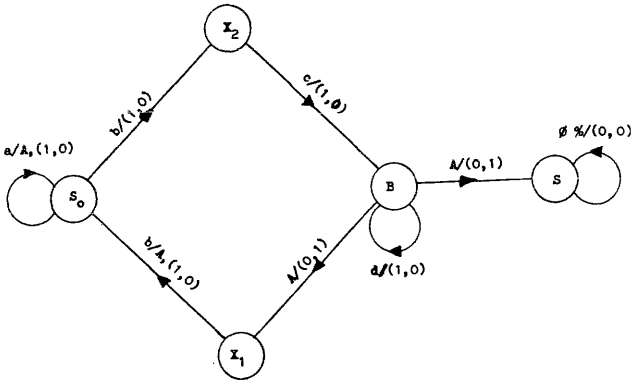


Figure 1.1.—The acceptor for the language of example 1.1

The acceptor of Figure 1.1 as shown is not deterministic in operation. This is because two possible transitions may occur from state B when symbol A is read on the PDS tape. In addition, a transition is defined for state B when symbol d appears on the machine input tape. To construct a deterministic version of Figure 1.1, we must construct a table listing a single transition for each possible pair of symbols on the input tape and pushdown store that can appear when the machine is in state B. We do so as follows:

(a) Consider a transition from state B to state S. In order that the input tape string be accepted, an A must appear on the left end of the PDS tape string, and symbolϕ (the boundary marker, indicating the end of the input string) on the input tape. The pair of symbols ϕA can be read in state B. For no other pair of symbols can a transition occur from B to S, since S will then be reading an undefined pair of symbols.

(b) Consider a transition on the loop from state B back to state B. This transition can occur whenever symbol d appears on the input tape and either A or % (the boundary marker of the PDS tape) is read on the left end of the PDS tape string. Since no other transition between states of the automaton involves reading symbol d on the input string, whenever symbol d is read on the input tape in state B, a transition occurs along the loop leading back to state B.

(c) Consider a transition from state B to state X_1. To make this transition, an A must be read from the PDS tape. In addition, only one transition (in this example) is defined from the state following B on this path, namely from X_1. Thus, whenever A appears on the PDS tape and symbol b is read from the input tape in state B, a transition occurs to X_1.

Cases (a), (b), and (c) represent all possibilities that appear in this example. Hence, all transitions from state B are uniquely defined in terms of pairs of symbols read from the input and PDS tapes, and we can construct the deterministic version of Figure 1.1, as shown in Figure 1.2.

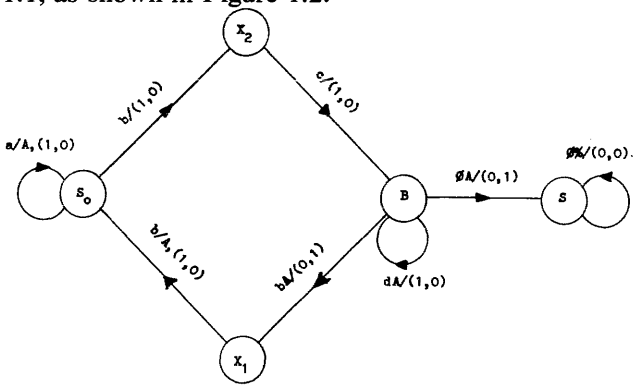


Figure 1.2.—The deterministic version of figure 1.1

In Example 1, all transitions possible between states of the PDS acceptor are listed. When the acceptor is performing a computation in which some symbol read from the input or PDS tape is not defined for any transition from the current state of the acceptor, the input string is not part of the language accepted by the machine. It happens that this property of undefined transitions is true in general for machines constructed by the algorithm illustrated above. Because of the correspondence between states of the PDS acceptor and rules of its grammar, undefined machine transitions during some computation of the machine automatically signal the source of errors in the imput string (or computer program) being processed. By our convention, the computation that accepts some string of a language must terminate in a state corresponding to the initial symbol of its grammar. In Example 1, the initial symbol is S, and the final state of the constructed machine is state S.

As a second example, consider the following grammar of Eickel *et al.*:[4]

Example 2. An ALGOL Sub-Grammar of Eickel.

$Z : := VcZ$ $\quad Z : := V$ $\quad V : := Ue$
$Z : := e$ $\quad U : := adb$

In the grammar above, there is a rule of the form

$$Z : := V \qquad (8$$

with both Z and V nonterminal symbols. Such a rule

cannot appear in a grammar converted into a PDS acceptor using the algorithm of this paper. Since a weakly equivalent grammar can always be constructed so that rules like (8) are absent,[1] we construct Grammar 2.1, weakly equivalent to Example 2 and lacking rule (8) above:

Grammar 2.1. A Grammar Weakly Equivalent to

Z : := VcZ Z : := Ue V : := Ue
Z : := e U : := adb

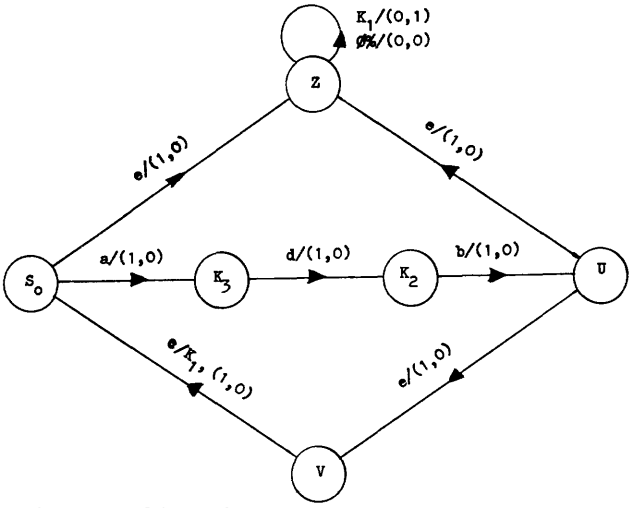


Figure 2.1.—The PDS acceptor constructed from grammar 2.1

The acceptor constructed from the normal-form version of Grammar 2.1 is shown in Figure 2.1 above. Because of the choice of this grammar, two transitions from state U are possible for the same input-tape symbol. Figure 2.2 indicates the transition from state U, and the transitions which can follow. As can be seen in Figure 2.2, the sequence of symbols ec

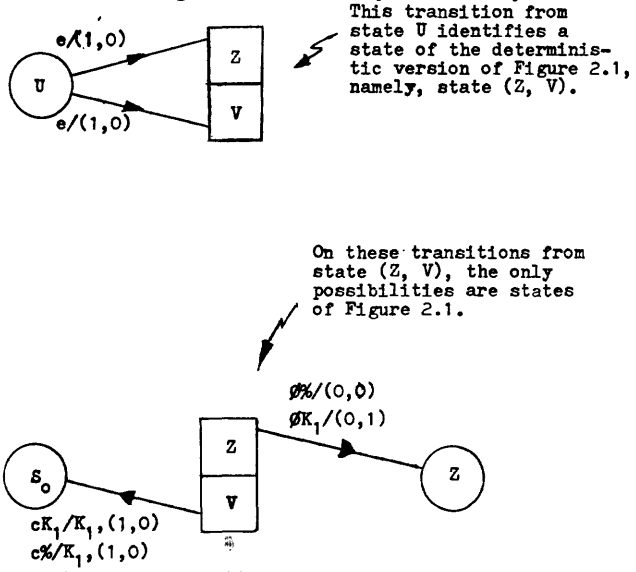


Figure 2.2.—Transitions following state U of the acceptor in figure 2.1

appearing on the input tape unambiguously causes a transfer to state S_0 from U by way of state (Z,V). The sequence of input tape symbols eϕ, with e read on the input tape in state U and ϕ read on the input tape in the combined state (Z, V) unambiguously causes a transfer from (Z, V) to Z. Using these transitions of the states following state U, we can immediately construct the deterministic version of Figure 2.1, as shown in Figure 2.3.

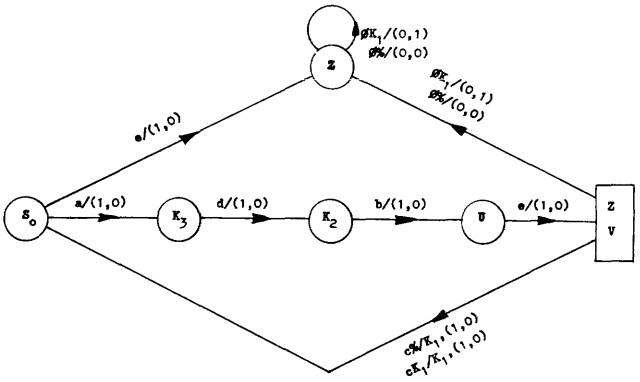


Figure 2.3.—The deterministic version of the acceptor in figure 2.1, incorporating the transitions of figure 2.2

Automatic construction of tree diagrams

As can be verified readily, the string of symbols (8)

s = abcbbcbbcd (8)

belongs to the language of Example 1. Using the criteria of Section III of this paper, we can indicate the correspondence between the record of a computation of the machine in Figure 1.2 given string (8) above, and the construction of the appropriate tree diagram over that string. Figure 3 below shows the sequence of state transitions made by the machine of Figure 1.2 when accepting string (8) on its input tape. The lines drawn between state symbols and input tape symbols in the figure can be produced automatically, since lines only connect states with symbols read by states or with immediately preceding states of the computation.* Circles are drawn around PDS symbols in Figure 3 to distinguish them from states. PDS tape symbols recognized during the computation are located where a state would be written in the diagram. Thus, initial state S_0 does not appear as a symbol on the constructed tree diagram.

ACKNOWLEDGMENT

The author wishes to acknowledge many valuable discussions on the subject of the paper with Drs. A. A. Grau, G. K. Krulee, and E. Shamir, all of Northwestern University.

REFERENCES

1 V SCHNEIDER
Pushdown-store processors of context-free languages
Doctoral Dissertation Northwestern University
Evanston Illinois 1966

*A more detailed account of this algorithm can be found in reference 1.

2 R W FLOYD
The syntax of programming languages-a survey
Comm ACM 6 451 1963
3 T V GRIFFITHS S R PETRICK
On the relative efficiencies of context-free grammar recognizers
Comm ACM 8 289 1965
4 J EICKEL M PAUL F L BAUER K SAMELSON
A syntax controlled generator of formal language processors
Comm ACM 6 451 1964
5 S GINSBURG S GREIBACH M HARRISON
Stack automata and compiling
System Development Corp Document No TM-738/021/00
Santa Monica 1965
6 G K KRULEE D M LANDI
Ambiguity and delay in finite encodings
Dept of Industrial Engineering and Management Science
Research Memorandum Northwestern University
Evanston Illinois 1965
7 N CHOMSKY
Formal properties of grammar
in Luce R Bush R and Galanter E Eds
Handbook of mathematical psychology
vol II pp 323-418 Wiley New York 1963
8 Y BAR-HILLEL M PERLES E SHAMIR
On formal properties of simple phrase structure grammars
Zeitschrift fuer Phonetik, Sprachwissenschaft und
Kommunikationsforschung 14 143 1961

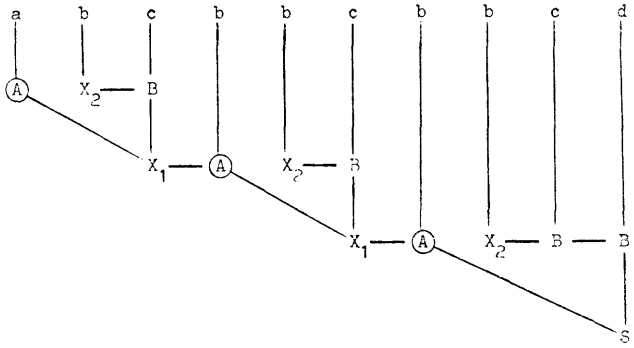


Figure 3.—A tree diagram record of the computation performed by the acceptor of figure 1.2 on string (8)

The design and implementation of a table driven compiler system

by C. L. LIU and G.D. CHANG*
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

R. E. MARKS
University of Illinois
Urbana, Illinois

INTRODUCTION

The broader application of digital computers to various areas of studies has prompted the design and usage of special purpose problem-oriented programming languages. Although designing and writing a compiler for a special purpose language is no more a mysterious task as it was ten years ago, it is still, in most cases, a very tedious task that might take a large number of man-months to perform. The purpose of designing a generalized table-driven compiler system is to allow a user to write his own compiler for his special language at a reduction of the time currently required in the implementation of a compiler for a new source language. It should be pointed out that not only can a user design a compiler of his own but he can also make modifications, large or slight, to an existing compiler developed with the system. This, therefore, also provides an ideal simulation environment in connection with the implementation of new ideas in translation process. This particular line of development in translation systems naturally lends itself to the increasing use of digital computers in a time-sharing environment.

The notion of a "table-driven compiler" is an extension of the notion of a "syntax-directed compiler" first studied by Irons.[1,2,3,4] The difference between a conventional compiler and a syntax-directed compiler is that in the former, the syntax of the source language is essentially buried in the coding of the compiler itself. The compiler is, therefore, rigidly bound to a fixed source language, and the slightest deviation from the original syntax is, of course, forbidden. In a syntax-directed compiler, the encodement of the syntax structure of the source language, usually in some form of tabular data structure, is separated from the other portions of the compiler and is used to control the operation of the compiler. It is, therefore, possible to replace the encodement of the syntax structure of one source language with that of another when the compiler is used to translate programs written in the other source language. The idea of using replaceable tables to control the operation of a compiler is extended in a table-driven compiler system. Besides having a syntax table to control the syntactic analysis, we shall also have tables to control the allocation of storage space as well as the assembly of binary machine codes. It follows that not only can we modify the syntax of the language that is to be translated by the compiler, but we can also specify the ways in which the compiler allocates storage space and generates object programs. Therefore, to design a compiler for a new source language, we have only to design these tables that control the correct operation of the compiler system. This is a much easier task than that of writing a new compiler for the source language.

General organization

In this paper, we describe a "Table-driven Compiler System" which is designed and implemented[5,6,7] on the IBM 7094 Computer. Our goal is to provide the users of the system with an environment within which they can freely design and produce their compilers. The primary design criterion is generality so that the users can define a large class of input languages oriented toward any kind of problem solving purposes, and can also define a large class of object programs to be executed on different computer systems. Therefore, in our system we do not limit the users to specific ways of doing syntactic analysis, or doing storage allocation, or producing binary pro-

grams of a specific format for a particular computer system. What we provide are mechanisms that are general enough for whichever way a user desires to build his compiler.

The Table-driven Compiler System consists of a base program, two fixed higher level languages: the Table Declaration and Manipulation Language and the Marco Interpretation Language together with the corresponding translators which generate the control tables according to the user's specification. A third higher level language: the Syntax Defining Language and its corresponding translator are also needed. However, their definitions are left to the users for the reason of providing them with greater flexibility in specifying the method of syntactic analysis. The base program is controlled by the control tables to perform the task of translating source programs into object machine codes. It is a general program which is independent of the particular source language being translated as well as the method of translation. The control tables contain an encodement of the syntax of the source language, an encodement of the method of translation and an encodement of the characteristics of the target machine.

The base program in the system is divided into three segments; the syntactic analyzer, the table processor, and the assembler. Each segment is controlled by one or more control tables. Their organization is shown schematically in Figure 1. The syntactic analyzer accepts programs written in the source language, recognized syntactic types, and transmits information that will be used for storage allocation to the table processor. After syntactic types have been recognized, the syntactic analyzer also generates

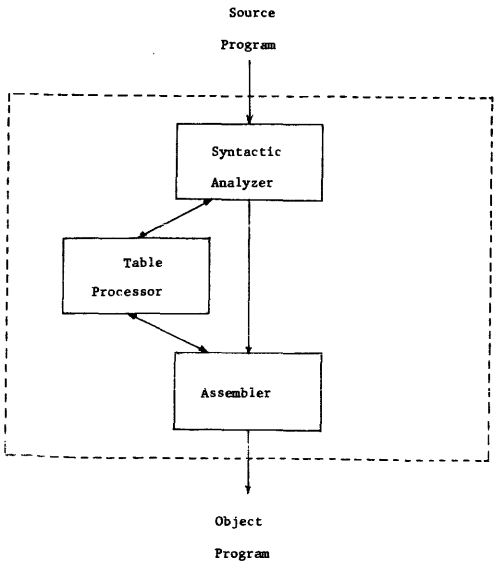


Figure 1 – Organization of the base program

a list of macros which will be interpreted by the assembler at assembly time. The syntactic analyzer is controlled by three tables, the lexical table, the test table, and the action table. The table processor can be divided into two parts. The first part accepts information from the syntactic analyzer and enters them into a set of tables called information tables. The second part manipulates all the information tables. After the entire source program is scanned by the syntactic analyzer, this part will sort, merge these tables, and ultimately assign addresses to all the symbols and literals defined in the program so that storage allocation information will become available to the assembler. The table processor is controlled by two tables: the main directory, and the table-manipulation control table. The assembler accepts the list of macros from the syntactic analyzer and generates the binary object program. The assembler is controlled by a macro interpretation table. In addition, the assembler uses another table which contains the binary representation of machine instructions, but is not controlled by it. Figure 2 is a schematic diagram showing how a source program is processed by the compiler system to produce the final object program.

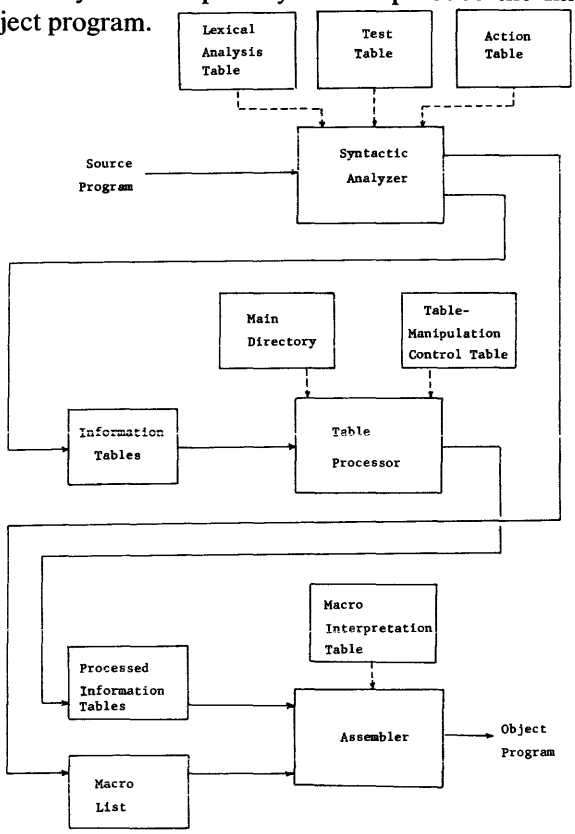


Figure 2 – Information flow in the system

To design a compiler for a particular source language, a user must prepare a set of control tables that are appropriate for this source language. Using the Syntax Defining Language, he shall specify the syntactic rules of the source language and the way macros are generated upon the recognition of syn-

tactic types. Using the Table Declaration and Manipulation Language, he shall declare all the information tables to be used in the compiler, and the way these tables are to be manipulated. Then, he must specify the meanings of the macros generated by the syntactic analyzer by using the Macro Interpretation Language. These specifications will all be assimilated by the respective translators and converted into different control tables.

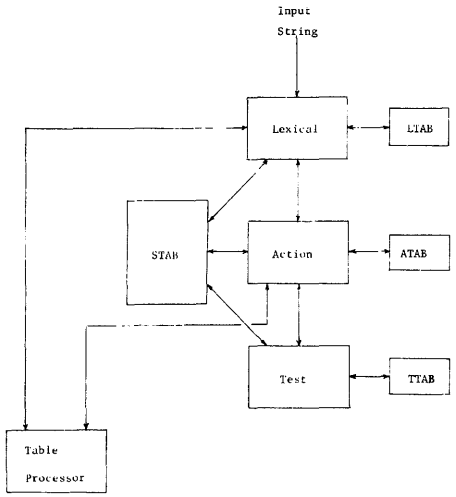


Figure 3 – Organization of the syntactic analyzer

The syntactic analyzer

The syntactic analyzer operates on the input string to generate a corresponding list of macros. It carries out a series of comparisons between sections of the input string and sections of the encodement of the syntax of the language. When a syntactic pattern is recognized, a set of operations will be performed.

The analyzer consists of three basic routines, called LEXICAL, TEST, and ACTION. These routines are controlled by control tables, respectively called LTAB (Lexical analysis TABle), TTAB (Test TABle), and ATAB (Action TABle). There is another table STAB (System TABle) which is used to store the intermediate results of the syntactic analysis and the system variables. Figure 3 shows schematically the organization of the syntactic analyzer.

Routine LEXICAL, controlled by LTAB, performs the lexical analysis on the input string. When a basic syntactic type (e.g. variable name, literal) is recognized, the result of lexical analysis will be put in one of the information tables in the table processor. A pointer, corresponding to this entry, is returned by the table processor to the syntactic analyzer. This pointer will be stored in the table STAB by the routine LEXICAL and is used to represent the entry in the information table throughout compilation.

Routine TEST performs the comparisons between the pointers in STAB or the table processor entries associated with the STAB pointer and an encodement of the syntax which is stored in the table TTAB. When a successful sequence of tests are performed, that is, when a certain syntactic pattern is found, control is changed to routine ACTION.

Routine ACTION has quite general arithmetic and system control facilities. ACTION, as controlled by the table ATAB, generates a list of macros by manipulating the pointers tested by routine TEST. ACTION is the only routine which can change the STAB-table processor structure; TEST cannot; and LEXICAL may only add entries to it in a very controlled manner. The list of macros will be interpreted by the assembler later to assemble binary machine codes. ACTION also performs many other bookkeeping operations upon the fields of the pointers. For example, a field in each of the entries in the symbol table may be used to keep track of identifier usage – whether a given identifier has been used before and if so, how and when, is its present usage consistent with previous usage (e.g. is a label now being used as an indexed array?). It is routine ACTION with table ATAB that would manipulate and test these fields.

The data structure

The table processor is external to the syntactic analyzer in the sense that all the communication between them are done via interface subroutines. Within the analyzer an entry in an information table in the table processor is represented by a pointer issued by the table processor. Within the table processor there are two values associated with this pointer – a location value and a table value. The table value identifies the information table in which the entry is stored. Moreover, the table value also gives the location within the table processor of a set of packing and fielding information which tells how the information associated with an entry is to be broken up into fields. The location value is the location within the table processor which contains the actual information associated with an entry. This block of information may be broken into many fields, each field giving a single characteristic of the entry. In practice, the size of a field may range from one bit to several computer words. It is intended that information tables such as SYMBOL TABLE, LITERAL TABLE, and TERMINAL SYMBOL TABLE are to be organized and used in the table processor. Even though the elements in these tables are thought of as entirely different objects, they are all represented in the same way – by a pointer – in the syntactic analyzer.

Inside the syntactic analyzer, all the results of the

analysis are stored in the table STAB. Two basic types of variable can be used in the analyzer—"value" and "pointer." A value simply has a numeric value and is treated as such. A pointer is either one that points to an entry in an information table or one that points to an entry in STAB. An entry in STAB may be a simple unindexed value or pointer, or it may be an indexed element in an array or an element in a pushdown stack. STAB may be organized into different number of pushdown stacks and arrays by the user. Such flexibility in organizing the table STAB is desirable as different methods of syntactic analysis might use different numbers of arrays and stacks.

The routine test

The routine TEST makes a series of comparisons between fields of pointers and numeric constants in order to identify syntactic types. Each comparison made by routine TEST is encoded in sixteen fields which require three to four words of 7094 storage. These fields are stored in table TTAB.

The tests performed in TEST are rather simple; if an extremely complex test is needed, control can be transferred to routine ACTION which operates more slowly than TEST but has a completely general arithmetic—comparison facility. Such a call to ACTION is in a different mode and should not be confused with the return to ACTION after a sequence of TESTing. It is termed a predicate call. The result of a predicate call is a truth value which upon return from ACTION is used in the same manner as is a truth value computed internally in TEST.

The routine action

Routine ACTION performs the various arithmetic and manipulative functions necessary for bookkeeping and for moving the pointers that represent the input elements into a list of macros. Thus TEST checks the input string for the existence of syntactic patterns; when a pattern is found, control is passed to routine ACTION. When ACTION completes its processing sequence, control is again transferred to TEST for more pattern testing. The instructions for ACTION are encoded in table ATAB. Part of an ATAB line either describes a location within the data structure or gives numeric values and part defines the operations ACTION perform.

The routine lexical

Lexical analysis is the process of analyzing the input string and putting the results of this analysis in the table processor. The essential results of lexical analysis are a pointer to the information table entry and a truth value indicating whether or not lexical analysis did in fact find an acceptable BCD sequence.

Lexical analysis is actually a test of syntactic patterns in the input string. However, due to the specialized nature of the task, it was thought that a significant decrease in operation time could be achieved if lexical analysis was isolated into a separate routine rather than being made a function of the more general TEST-ACTION syntactic analysis. If LEXICAL finds—according to the rules in LTAB—an acceptable pattern, LTAB indicates that either a search of a certain information table in the table processor is required or by the nature of the string it is the BCD representation corresponding to a certain entry reference number and no search is required. An example of this second case is when LEXICAL is checking for specific terminal symbols such as "+", "−", or "*". It is possible that both the table processor and LTAB are initialized in such a way that the correct entry reference number for each terminal symbol is encoded into LTAB, and thus when the specific terminal symbol is discovered no table processor search is required.

LTAB is divided into many blocks each one of which contains the analysis information for one syntactic type. In general LEXICAL is called by a sequence of several ATAB lines each one pointing to a different LTAB block; LEXICAL performs the analysis described in each block in order of occurrence in ATAB. If a block of analysis fails, LEXICAL automatically—without returning to ACTION—performs the lexical analysis pointed to by the next ATAB line. When the analysis succeeds, the pointer is formed and placed in STAB, the system truth variable is set to TRUE; the ATAB index is cycled down to beyond the last LEXICAL operation; and control is returned to ACTION. If the block of analysis pointed to by the last ATAB line in the LEXICAL sequence fails, the system truth variable is set to FALSE; and control is returned to ACTION.

The table processor

When doing the analysis, the syntactic analyzer picks up information that should be organized, processed, and made available later on to the assembler. For example, all the variable names appearing in a program should be collected and storage registers be allocated to them before binary machine codes can be assembled. The allocation of storage registers requires information such as whether a variable is an array, the size of the array, and the dimensions of the array, and so on. Unlike the generation of macros which can be carried out when the syntactic analyzer has recognized a syntactic type that is "large" enough (for example, a statement) to warrant a generation, these information can be processed only at the end of the syntactic analysis phase. The table

processor is designed for the collection and processing of information such as these. It works together with the syntactic analyzer to enter useful information to various information tables, and processes these tables after the syntactic analyzer has completed the analysis.

The information tables

To design a compiler for a particular source language, the user can have in the compiler any number of information tables. These information tables are used to store different symbols, labels, literals, integers, character patterns, dimensional information, and other information that the syntactic analyzer has scanned in the source program. The number of information tables as well as the size and format of each table are declared by the user.

An information table has a very simple structure. Each entry to an information table contains a certain number of fields and occupies a certain number of registers. The detailed format description of the information tables are stored in the main directory. In the table, there is a bookkeeping word containing two pieces of information: the current top of the table which is defined as the register above which there is no other entries, and the current entry of the table which is defined as the entry being processed by a utility routine. The information on the current top of the table is needed when a new entry is added to an unsorted table or when a table is to be sorted. The information on the current entry of the table is used by the search or insert routine to indicate that a match or a place for insertion has been found.

The main directory

The format description of the information tables is stored in a table called the main directory. It is clear that the table processor has to go through the main directory in order to make meaningful accesses to the various information tables. The format information of each information table occupies a block of registers in the main directory.

The maximal number of entries of an information table, as well as the number of fields in each entry are all declared by the user, using the Table Declaration and Manipulation Language. Using the Table Declaration and Manipulation Language, the user shall declare for each information table:

(1) the symbolic name,
(2) the size,
(3) the sorting option, that is, whether the table should be kept sorted all the time. If so, the field based on which the table is to be sorted and the sorting scheme (the order of precedence) the sorting routine should follow.
(4) the number of fields in an entry and the way they are packed.

The main pointer table

As mentioned before, when the syntactic analyzer calls the table processor to enter a piece of information into one of the information tables, a pointer will be returned to the syntactic analyzer. This pointer is linked to the entry in the information table through an entry in the main pointer table. It points to an entry in the main pointer table which in turn points to the entry in the information table. In each of the entries in the information table, there is a field which contains a pointer pointing back to its corresponding entry in the main pointer table. With the two-way pointers between the entries in the main pointer table and the entries in the information table, the linkage between a pointer in the syntactic analyzer and its corresponding entry in an information table will be maintained even after the information table is sorted or merged with another information table.

Besides pointing to an entry in an information table, an entry in the main pointer table also contains the name of the information table. In this way, format description of the information table can be looked up in the main directory. Every time an entry in an information table is displaced, the two-way pointer between the entry and its corresponding entry in the main pointer table will be updated.

Table manipulation

At the end of the syntactic analysis phase, the information tables set up in the system will be processed by the table processor. At this point, the source program has been scanned once; all the macros were generated by the syntactic analyzer, and all the useful information in the source program was entered into the information tables.

Before passing control to the assembler which generates the binary object program, we still must determine the storage locations to be assigned to the symbols, arrays, and literals. In addition, we also need to organize the information tables so that the assembler can have quick access to all the necessary information. For example, the assembler will have to know if a variable is defined to be an integer or a floating point number, or if the variable is an array name or just a simple variable, and so on. For this purpose, we want to group all the information together and possibly combine some of the information tables. These are the functions of the table processor.

The assembler

The assembler accepts the list of macros generated by the syntactic analyzer and uses the information tables furnished by the table processor to generate

binary machine codes. The assembler is controlled by the macro interpretation table which contains information on the meanings of the macros. The assembler uses, but is not controlled by, another table which contains the binary as well as the BCD representation of the machine instructions of the target machine.

The macro list

A macro generated by the syntactic analyzer contains the following information:

1. The macro name, which is actually a number by which the macro is referred to in the macro interpretation table.
2. A count, which is the number of times the result of this macro will be referred to by succeeding macros.
3. The list of arguments of the macro with the type of each argument appropriately identified. There are three types of arguments:

A type 0 argument is an entry in an information table. It is the pointer of the information table entry.

A type 1 argument is the result of a preceding macro in the list. It is a pointer to that particular macr.

A type 2 argument is a number.

A macro is stored in a block of registers with each argument occupying one register. There is also a blank register in the block, called the association register, the usage of which will be explained in the following:

The macro interpretation table

For each of the macros in the macro list, the assembler will generate the corresponding binary machine codes using information in the macro interpretation table. In general, a macro may be interpreted along different paths to yield different sets of binary codes. For example, depending on the modes of the arguments, whether the results of the preceding macros were left in some of the active registers, whether the result of the macro being interpreted will be referred to by succeeding macros and so on, the binary codes generated by the assembler for the same macro will be different. In the macro interpretation table, the way in which the macros are to be interpreted are stored. Also, in the macro interpretation table, information on the usage of temporary storage registers, the usage of active registers and error comments are also available for each of the macros.

Although we have chosen a fixed format for the macros in the system, flexibility is retained by the use of the macro interpretation table. It is interesting to observe that for the same source language and the same set of macros, different macro interpretation tables can be used. One might want to choose an interpretation table that generates highly efficient machine codes at the expense of compilation time, or one might want to choose an interpretation table that will give fast compilation but generates less efficient machine codes. Similarly, by changing the table containing the binary representation of the machine instructions, we can, from the same macro list, generate object programs for different target machines.

Temporary storage pool

When an algebraic expression or Boolean expression is being evaluated, it is necessary, in most cases, to save the intermediate results of computation in temporary storage registers. When the intermediate results are no longer needed, the registers used to save these intermediates may be freed and used in some other computations later on. The maximal number of temporary storage registers that can be used in the compiler is declared by the user. These temporary storage registers may be regarded as a pool from which registers can be requested and to which registers are returned.

Because there is no way of knowing the length of a program or the total number of registers used as temporary storage within the program until all the binary machine codes are generated, the block of temporary storage will have to be attached to the end of an object program. This means that the addresses of the registers used as temporary storage cannot be assigned until the machine instructions are all generated. Therefore, the assembler shall leave the addresses of temporary storage registers as floating addresses starting from zero in the course of compilation, and return to add to these floating addresses the "program break" at the end of the generation of all the machine codes. In order to identify these floating addresses in the machine instructions identification bits are attached to each instruction.

The temporary storage pool is organized as a chained list whose size is initialized by the user. When a temporary storage register is called for, the first register in the list of available registers will be used and thus be deleted from the chain. Whenever a temporary storage register is released, it will be put back in the pool and is placed at the beginning of the list of available temporary storage.

Active registers usage

The execution of a machine instruction would invariably involve the usage of one or more active registers. The register used may be the accumulator, or the multiplier-quotient register, or an index register. Keeping track of the contents of various active registers, the assembler can generate more efficient binary codes by the removal of redundant machine

instructions. For this pupose, an association list is set up in the assembler. For each active register, an entry is reserved in the list. If the execution of the instructions. For this purpose, an association list is in a certain active register, a two-way pointer will be set up between the corresponding entry in the association list and association register in the macro block. When an active register is used later on by some other computations, the two-way pointer will be erased. It is, therefore, necessary to include information on the active registers a macro will use and the active register in which the result of the macro is left in the macro interpretation table. With this information, the two-way pointer will be set up and erased correctly.

CONCLUSION

Besides providing the users with an environment in which they can write their own compilers, it is also hoped that the experience of designing such a system will lead to the understanding of the general theory of compiler structure and the general technique of compiler writing. We emphasize, in our design, the segmentation of the system so that the functions of each section will be clearly defined and be brought out in evidence. The communication problem between the segments is not a difficult one to handle as illustrated in our design. It should also be pointed out that for the generality and flexibility we try to attain, less consideration is placed on efficiency. A case in point is the general structure for the information tables in the table processor.

REFERENCES

1 E T IRONS
The structure and use of the syntax directed compiler
Annual Review in Automatic Programming Vol 3 1963

2 E T IRONS
A syntax-directed compiler for Algol-60
Comm ACM 4 1961

3 T E CHEATHAM K SATTLEY
Syntax-directed compiling
Proceedings SJCC Spring 1964

4 S WARSHALL R M SHAPIRO
A general purpose table driven compiler
Proceedings SJCC Spring 1964

5 G D CHANG
A table driven compiler generator system
S M Thesis M I T June 1966

6 R E MARKS
A table driven syntactic analyzer
S M Thesis MIT August 1966

7 C L LIU G D CHANG R E MARKS
The design and implementation of a table driven compiler system
Technical Report Project MAC to be published

A complex logic module for the synthesis of combinational switching circuits.

by YALE N. PATT
Cornell University
Ithaca, New York

INTRODUCTION

Continuing achievements in device technology (i.e., integrated circuits) are drastically changing the costs associated with the circuit realization of a switching function. Already the integrated circuit module has replaced the discrete transistor as the basic unit in a switching circuit. The factors which most greatly affect the cost of the integrated circuit module (i.e., the individual process steps, the packaging, the member of external interconnections, and the circuit yield) are almost wholly independent of the number of transistors on the monolithic chip contained within the module* Furthermore, a large percentage of these costs are independent of the number of modules manufactured; therefore, the more uses that can be found for a module, the less expensive the per unit cost of the module will be.

In view of the above, it is suggested that a valid measure of a switching circuit's cost is the number of modules within the circuit. If each module has the same number of pins then this cost function is ālso proportional to the number of interconnections in the circuit. In general this cost would be minimized if (1) all modules within the circuit were identical, and (2) the function realized by this "building block" module were chosen so as to minimize (on the average) the number of modules required to synthesize any arbitrary switching function.

This paper considers the problem of selecting and using a building-block module. First certain properties of switching functions are studied so as to determine what consitutes a good module. A somewhat heuristic set of criteria is established, and a module satisfying these criteria is chosen. Second, certain experimental evidence which supports this choice is presented. Finally, the implementation of this module in actual circuit synthesis is discussed.

Properties

This section discusses three properties which, it is argued, are desirable in a switching function if it is to be used as a building-block module: completeness, total asymmetry, and logical versatility. Simple tests are established which quickly determine whether or not a candidate function possesses these properties.

Completeness

A switching function f is logically complete if and only if any arbitrary switching function g can be expressed as a logical formula, using f as the only logical connective. In other words, if g is *any* desired function and f is the output of the only available building block module then g can *always* be synthesized if and only if f is complete.

Clearly, if we are to limit ourselves to one module, it must possess this property. Furthermore, it should possess this property in the strictest sense, that is, whether or not constant signals (0's, or 1's, B^+ or ground are admissible inputs to a circuit. This restriction is imposed in order to encompass those realizations (e.g., integrated arrays) wherein constant signals cannot be applied at zero cost.

To determine if a function is complete, we first establish necessary and sufficient conditions for completeness.

Definition 1. An input vector η_i is an n-dimensional binary vector, corresponding to a specific input combination in the domain of switching function f.

There are 2^n such input vectors, varying from 00 . . . 0 to 11 . . . 1. Note that the *jth* component of η_i (η_{ij}) corresponds to the value of x_j in row i of an n-variable truth table. Two input vectors η_i and η_j are

*There may be some argument with respect to yield, but I think it is generally agreed that this statement is valid within certain circuit-complexity bounds, which you will find are not exceeded in this study.

said to be *complementary* if $i + j = 2^n - 1$. Note that if $\eta_i = \bar{e}_1\bar{e}_2 \ldots e_n$, then $\eta_j = e_1e_2 \ldots \bar{e}_n$

Theorem 1. A switching function f is complete if and only if

(1.) $f(\eta_0) = 1$

(2.) $f(\eta_{2^n-1}) = 0$

(3.) $f(\eta_i) = f(\eta_j)$, for some i, j such that $i + j = 2^n-1$.

Proof of Theorem 1.

Necessity

Assume f is a switching function such that $f(\eta_0) = 0$. Consider any circuit constructed solely with f modules. Let each input variable equal 0. Any module in the circuit which obtains its input signals directly from these external inputs must have an output 0. Then any module which obtains its input signals from some combination of external inputs and outputs of preceding modules must also have an output 0. Then the output of this circuit g under the input combination η_0 is always 0. No function g such that $g(\eta_0 = 1$ can be realized solely with f modules. Consequently, f is not complete. Therefore $f(\eta_0) = 1$ is a necessary property.

The necessity of $f(\eta_{2^n-1}) = 0$ can be shown by a similar argument.

Consider a switching function that exhibits properties 1 and 2, but not property 3; its truth table is shown in Figure 1. Consider any pair of complementary

$x_1x_2 \cdots x_{n-1}x_n$	f
0 0 ... 0 0	1
0 0 ... 0 1	e_1
0 0 ... 1 0	e_2
0 0 ... 1 1	e_3
⋮	⋮
1 1 ... 0 0	$\bar{e}_3$
1 1 ... 0 1	$\bar{e}_2$
1 1 ... 1 0	$\bar{e}_1$
1 1 ... 1 1	0

Figure 1—Generalized truth table for a switching function which satisfies conditions 1 and 2 but not 3 of Theorem 1

rows i and j. If the inputs to a module f_1 are selected from the set $(x_1, x_2, \ldots, x_n)$, then, corresponding to each pair of complementary input combinations to the circuit. is a pair of complementary input combinations to the module. Such a module, accordingly, has complementary outputs for this pair of rows. Therefore, any succeeding module obtains its inputs from the set $(x_1, x_2, \ldots, x_n, f_1)$ with the same results. One can never devise a circuit for which $f(\eta_i) = f(\eta_j)$. Therefore, property 3 is necessary.

Sufficiency

Consider a function that has the three necessary properties. Select any η_i, η_j such that $f(\eta_i) = f(\eta_j)$, $i + j = 2_n-1$. Permute the input variables such that all variables having the same value in η_i (0 or 1) are lumped together. The switching function then has the truth table shown in Figure 2.

$x_{i_1} \cdots x_{i_k}$	$x_{i_{k+1}} \cdots x_{i_n}$	f
0 ... 0	0 ... 0	1
⋮		⋮
0 ... 0	1 ... 1	e_i
⋮		⋮
1 ... 1	0 ... 0	e_i
⋮		⋮
1 ... 1	1 ... 1	0

Figure 2—Generalized truth table for a complete function

Connect inputs $x_{i_1}, \ldots, x_{j_k}$ together as variable y and $x_{i_{k+1}} \ldots, x_{i_n}$ as variable z. The function $\overline{yz}$ is obtained if $e_i = 0$, or $\bar{y} + \bar{z}$ if $e_i = 1$. It can be shown easily that both of these functions are logically complete. Therefore, f is complete. Q.E.D.

The above theorem is of immediate value. If a potential cadidate function does not satisfy the three completeness criteria (this can be checked very quickly), then it cannot be used as the building block module, and may be eliminated from further consideration.

If C(n) is defined as the number of complete functions of n variables, it can be shown easily that

$$C(n) = 2^{2(2^{n-1}-1)} - 2^{(2^{n-1}-1)}.$$

Also,

$$\lim_{n\to\infty} \frac{C(n)}{2^{2^n}} = \frac{1}{4}.$$

In fact, even for $n = 4$, $\frac{C(n)}{2^{2^n}} = \frac{1}{4} - \frac{1}{2^9}$. Therefore, for practically all values of n, almost ¼ of all switching functions satisfy the completeness requirement.

Total asymmetry

A switching function f is said to be totally asymmetric if every permutation of its input variables results in a different output function. Since there are n!

possible permutations of the n input variables, a totally asymmetric function module is one that can realize n! different functions of n variables, depending on the order in which the n variables are applied. It is conjectured that a desirable property for a building block module is that of total asymmetry.

The reasons for this conjecture are best explained by extending an experiment by Hellerman.[2] Hellerman wanted to synthesize every function of three variables with the minimum number of three-input NAND gates. His method was to systematically compute the output of every possible circuit consisting of one, then two, then three, etc. modules. Each time he obtained a function which he had not obtained previously, he designated the corresponding circuit as the minimal realization of that function. Because of the exhaustive nature of this procedure, his results were, in fact, minimal.

Suppose the same procedure is used to obtain the minimal circuit realizations for all functions of some arbitrary number of variables using some complete switching function f as the sole building block module. Consider the advantages of f being totally asymmetric. If f is *symmetric,* then the output of a circuit is unchanged if the inputs of any or all of the modules within the circuit are permuted. Only *one* function of k variables is synthesized by the entire set of circuits obtained by these permutations. If, on the other hand, f is totally asymmetric, every permutation of the inputs to a given module changes the output of that module. It is therefore argued (without proof) that in such a case, the set of circuits obtained by these permutations gives a relatively larger number of different functions of k variables as outputs than could be obtained if f were not totally asymmetric. If more functions are obtained with a small number of modules, fewer are left to be synthesized as r increases. Consequently, the total number of modules needed to synthesize all functions of k variables would be less.

A test for total asymmetry will now be developed. Certain intermediate results are stated, but not proved. Complete proofs are contained in reference 1.

Definition 2. A transformation t is an operation on switching function f such that the input variables of f are permuted. The symbol

$$t_{(i_1 i_2, \, , \, i_m)(j_1 j_2 \, \ldots \, j_r) \, \ldots \, (k_1 k_2 \, \ldots \, k_s)}(f)$$

denotes the switching function which is obtained by permuting the input variables of f according to the following patter

$$\begin{pmatrix} i_1 i_2 & \ldots & i_{m-1} i_m \\ i_2 i_3 & \ldots & i_m \; i_1 \end{pmatrix} \begin{pmatrix} j_1 j_2 & \ldots & j_{r-1} j_r \\ j_2 j_3 & \ldots & j_r \; j_1 \end{pmatrix} \begin{pmatrix} k_1 k_2 & \ldots & k_{s-1} k_s \\ k_2 k_3 & \ldots & k_s \; k_1 \end{pmatrix}$$

The set of transformations T form a group.

Definition 3. Two switching functions f_1 and f_2 are *equivalent* if there exists a $t \epsilon T$ such that $t(f_1) = f_2$.

It can be shown that the relation is an equivalence relation and as such partitions the set of all switching functions into disjoint subsets, or equivalence classes. Each equivalence class represents the set of functions which can be realized with a single module by simply permuting the correct input variables. The equivalence class of f is denoted T(f). If f is totally asymmetric, there are n! elements in T(f).

Lemma 1.* The number of switching functions in T(f) is equal to n! divided by the number of transformations $t \epsilon T$ which leave f invariant.

Lemma 1 states that to determine if the n! transformations of f are all distinct, one need not make $\frac{(n!)^2}{2}$ comparisons. In fact, it is necessary only to compare each of the n! functions t(f) to f. This comparison is facilitated by the use of the irredundant ring sum expansion of f.

Definition 4. A ring sum expansion is an exclusive-OR summation of terms, each of which is the AND function of some subset of the uncomplemented input variables. A ring-sum is said to be irredundant if each term in the summation occurs only once.

For every switching function there exists a unique irredundant ring sum expansion. To test a switching function for total asymmetry, it must first be expressed as an irredundant ring-sum. This is accomplished as follows:

(1) Express the function in disjunctive normal form. Replace each inclusive-OR symbol by an exclusive-OR.
(2) Replace every complemented variable $\bar{x}_i$ by $(1 \oplus x_i)$.
(3) Perform any necessary multiplication to remove all parentheses.
(4) Remove all pairs of identical terms. The resulting function is the irredundant ring sum expansion of f.

The fact that these four steps do not change the truth value of the function can be derived from the following statements:

(1) if α_i and α_j are minterms, then $\alpha_i \alpha_j = 0$.
$$\therefore \alpha_i + \alpha_j = \alpha_i \oplus \alpha_j$$
(2) $\bar{x} = 1 \oplus x_i$.
(3) $x_i(x_j \oplus x_k) = x_i x_j \oplus x_i x_k$
(4) $f_i \oplus f_i = 0$.

The terms of the ring sum expansion are partitioned into sets such that all terms in the same set have the same number of uncomplemented input variables. Label the sets $1_1, 2_1, \ldots k_1$.

Definition 5. A term vector β_i^r is a k_r dimensional vector such that β_{ij}^r equals the number of terms in the

*This is lemma 2.2 in reference 1.

partitioned set j_r (defined above) which contains variable x_i.

Definition 6. Two variables x_i and x_j are pseudo-equivalent with respect to r if their corresponding term vectors β_i^r and β_j^r are equal.

Further partition the terms of the ring sum expansion such that two terms are placed in the same subset if and only if their corresponding variables are pseudo-equivalent. Relabel the partitioned sets $1_2, 2_2, \ldots k_2$. Iterate this process until either no further partitioning is possible, or there are no pseudo-equivalent variables. Note that two variables can be pseudo-equivalent after iteration m(i.e., $\beta_i^m = \beta_j^m$), but no longer pseudo-equivalent after iteration r, r > m (i.e., $\beta_i^r \neq \beta_j^r$).

Theorem 2. If there exists an r such that $\beta_i^r \neq \beta_k^r$ for all $i \neq k$ (i.e., no pseudo-equivalent variables), then the switching function is totally asymmetric.

The proof of theorem 2 follows very closely the proof of theorem 3.9 of reference 1.

Note that the implication in theorem 2 is in one direction only. There are functions (although, fortunately, only a relatively small number of them) which are totally asymmetric, but which have $\beta_i^r = \beta_k^r$, for some $i \neq k$, for all r. One example is $f = x_1\bar{x}_2\bar{x}_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_4x_5 + \bar{x}_2x_3\bar{x}_5x_6 + \bar{x}_1x_2x_5\bar{x}_6 + x_1\bar{x}_3x_4\bar{x}_5 + \bar{x}_1x_3x_4\bar{x}_5x_6 + x_1x_2x_3\bar{x}_4\bar{x}_5\bar{x}_6 + x_1x_2x_3x_4x_5x_6$, which have the following term vectors:

$$\beta_1^r = \beta_2^r = \beta_3^r = (1,2,2)$$
$$\beta_4^r = \beta_5^r = \beta_6^r = (0,1,2)$$

Consequently, if the iterated procedure does not result in all unique term vectors, a final test for asymmetry must still be made.

Lemma 2.* Let t* represent any transformation in which all its permutation cycles are equal and prime. Then, if $t^*(f) \neq f$, $t^* \epsilon T$, there exists no $t \epsilon T$ (except the identity element) such that $t(f) = f$.

Lemma 2 reduces the number of comparisons further. The function f need only be compared to functions of the form t*(f). More than that, only those t* need be considered whose permutation cycles consist of pseudo-equivalent variables. If there exists no t* from that set such that $t^*f = f$, the function is totally asymmetric. If there exists a t* such that $t^*(f) = f$, then f is not totally assymmetric.

The discussion of total asymmetry concludes with an example of this test procedure.

Example: Test $f = x_1x_3 + x_2x_3 + x_1x_2x_4 + x_1\bar{x}_2x_4 + x_1x_2\bar{x}_4$ for total asymmetry.

Solution:

(0.) Express f as an irredundant ring sum.

$f = x_1 \oplus x_x \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_4 \oplus x_2x_4 \oplus x_1x_2x_4 \oplus x_1x_3x_4$.

*This is corollary 3.2 in reference 1.

(1.) Partition the terms of f such that terms in the same subset have the same number of variables.

$1_1 = (x_1,x_2,x_3)$
$2_1 = (x_1x_4, x_2x_4, x_1x_3, x_2x_3)$
$3_1 = (x_1x_2x_4, x_1x_3x_4)$

Form the β_i^1 vectors. (Note that at this point x_1, x_2, and x_3 are pseudo-equivalent)

$\beta_1^1 = \beta_2^1 = \beta_3^1 = (1,2,2)$
$B_4^1 = (0,2,2)$

(2.) Further partition the terms of f such that two terms are in the same subset if and only if their corresponding variables are pseudo-equivalent.

$1_2 = (x_1,x_2,x_3)$ $\quad 3_2 = (x_1x_3,x_2x_3)$
$2_2 = (x_1x_4,x_2x_4)$ $\quad 4_2 = (x_1x_2x_4,x_1x_3x_4)$

Form the β_i^2 vectors.

$\beta_1^2 = (1,1,1,2)$ $\quad \beta_3^2 = (1,0,2,1)$
$\beta_1^2 = (1,1,1,1)$ $\quad \beta_4^2 = (0,2,0,2)$

(3.) The β_i^2 vectors are distinct. Therefore, f is totally asymmetric.

Logical versatility

It is conjectured that a function's value as a building block module depends to some extent on the kinds of different functions which can be obtained from a single module by judiciously biasing and/or duplicating the inputs of that module.

Definition 7. A function f of k variables is a subfunction of g of n variables (n > k) if f can be obtained from g by biasing and/or duplicating one or more of the input variables of g.

Recall the extended experiment of Hellerman discussed in the previous section. All functions of k variables are to be realized by a minimal number of modules by systematically computing the output function of all circuits of one, then two, then three, etc. modules. Consider the effects of the kind of subfunctions contained in f on the minimal circuits obtained. It is suggested that the greater the variation available in the outputs of each module in a circuit, the larger the number of functions which can be obtained at the output of that circuit. Consequently, fewer functions would be left to be realized with more costly circuits, making the building block module more valuable.

Note that this property has been stated as "the greater the variation in the subfunctions of a module." No success has been achieved in developing a quantitative measure of this property. It is clear that simply counting the number of subfunctions is not enough, since some subfunctions add practically nothing to a function's logical capability while others add a great deal. For example, compare two subfunctions of g_1, $f_1 = x_1 \oplus x_2 \oplus x_3$ and $f_2 = x_1x_2\bar{x}_3$, with two subfunctions of g_2, $f_3 = x_1x_2x_3$ and $f_4 = \bar{x}_1\bar{x}_2x_3$. It ap-

pears that g_1 has a good deal more logical capability. Until some better measure is established, a potential module is qualitatively evaluated as one with a good deal of logical versatility or one with relatively very little.

The WOS module

One function which possesses the three properties discussed in section II is the WOS module.

Definition R. The WOS module of n variables is an n-input, one output circuit which realizes the following switching functions:

$$f = 1 \oplus x_1 \oplus x_3 \oplus x_4 \oplus \ldots \oplus x_n \oplus x_1x_2 \oplus x_1x_2x_3 \oplus \ldots \oplus x_1x_2 \ldots x_{n-1}$$

The significance of the letters WOS is discussed in detail in reference 1; it is not important to the understanding of this paper.

First let us show that for all n, the WOS module is complete, totally asymmetric and has a good deal of logical versatility.

The WOS module is complete since it satisfies the three conditions of theorem 1.

(1) $f(\eta_0) = 1 \oplus 0 \oplus 0 \oplus \ldots \oplus 0 = 1$.

(2) $f(\eta_{2^n-1}) = 1 \oplus 1 \oplus \ldots \oplus 1$. Since there are an even number of terms (2^n-2) in the ring sum expansion, $f(\eta_{2^n-1}) = 0$.

(3) $f(000\ldots010) = 0$, since two of the terms in the expansion equal 1.
$f(111\ldots101) = 0$, since all but two of the (2n-2) terms equal 1.

The WOS module is totally asymmetric since its β_i^1 vectors are all distinct.

$$\beta_1^1 = 1,1,1,1 - 1$$
$$\beta_2^1 = 0,1,1,1 - 1$$
$$\beta_3^1 = 1,0,1,1 - 1$$
$$\beta_4^{11} = 1,0,0,1 - 1$$
$$\beta_n^1 = 1,0,0,0 - 0$$

The WOS module has a good deal of logical versatility. Note, for example, that

$$f(x_1,0,x_3, \ldots x_n) = 1 \oplus x_1 \oplus x_3 \oplus x_4 \oplus \ldots \oplus x_n,$$

the exclusive-OR of n-1 variables. Also, for n even,

$$f(x_1,x_1,x_2,x_2, \ldots x_{\frac{n}{2}}) = x_1x_2x_3 \ldots x_{\frac{n}{2}},$$

the AND of $\frac{n}{2}$ variables. For n odd, the AND of $\frac{n+1}{2}$ variables is a subfunction of f.

A procedure has been developed which utilizes this n-variable module to synthesize any arbitrary combinational switching function in two levels with a minimum number of modules. It can be shown that on the average significant savings result if switching functions are implemented with this module, compared to implementation with two levels of AND/OR (or NAND/NAND) logic.

This paper considers a special case of the WOS module; i.e., n = 3.

$$f(x_1,x_2,x_3) = 1 \oplus x_1 \oplus x_3 \oplus x_1x_2.$$

This three-input module is a handy building block which can be used for designing non-repetitive logic circuits. It is also small enough to be considered as the basic cell in an integrated array.

An experiment

This section describes an experiment which measures to some degree the relative effectiveness of the three-variable WOS module. The 256 functions of three variables were synthesized with a minimal number of WOS modules. These circuit realizations were then compared with the minimal circuit realization of all functions of three variables using the three-input NAND gate as the basic module.[2]

Minimality was guaranteed by using the exhaustive technique discussed in section II. A circuit was chosen as the minimal realization of f if it consisted of r modules such that no circuit consisting of s modules, $s < r$, exists which realizes f. Since every function can be realized with at most three modules, this necessitated computing the output of every circuit of two modules. These were computed by substituting elements of the set {1,0,a,b,c} into the expressions for f_1,f_2, and f_3 shown in Figure 3.

The table below compares the two sets of circuit realizations.

Module	Fan in	Fan-out	Total Modules Req'd	Average per ckt.
NAND	3	3	1129	4.41
WOS	3	1	524	2.04

Implementation of the WOS module

No optimal synthesis procedure (minimal number of modules and therefore pins) exists for the three variable WOS module. In fact, no such procedure exists for the less complex AND/OR logic either. (This is the well known unsolved factoring problem.) A few guidelines, however, should lead the logic designer to a reasonably good circuit for any function.

The method is to systematically decompose the desired output function by means of the WOS module into functions of fewer and fewer variables, until only uncomplemented input variables and constants are required as inputs to the circuit. The following procedure is suggested:

(1.) Represent the function as an irredundant ring sum expansion.

(2.) Factor this expression into the form $f_1f_2 \oplus f_3$, such that the variables present in f_1,f_2, and f_3 are *as nearly disjoint as possible* and that the

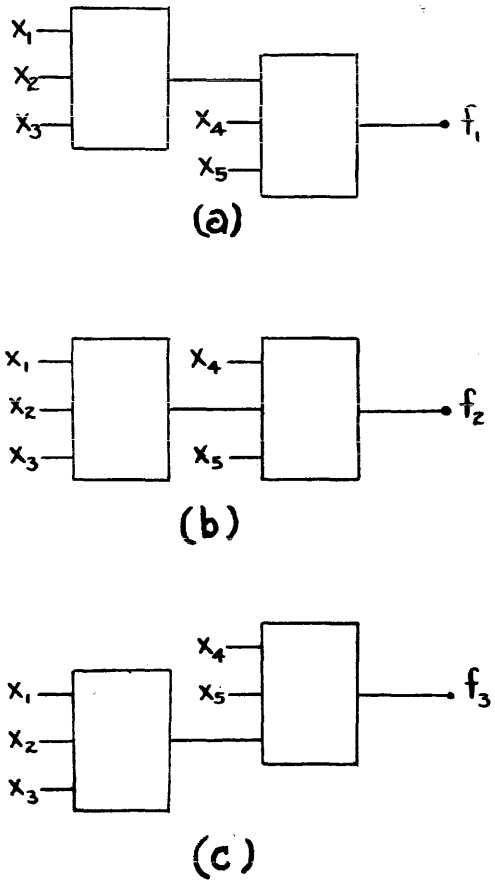


Figure 3—All circuits which consist of two three-input WOS modules

(a) $f_1 = x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_1x_2 \oplus x_1x_4 \oplus x_3x_4 \oplus x_1x_2x_4$

(b) $f_2 = 1 \oplus x_5 \oplus x_1x_4 \oplus x_3x_4 \oplus x_1x_2x_4$

(c) $f_3 = x_1 \oplus x_3 \oplus x_4 \oplus x_1x_2 \oplus x_4x_5$

maximum number of variables in any one of the functions is *as small as possible.*

(3.) Iterate this procedure for f_1,f_2, and f_3.

Note that one module decomposes f into three functions of fewer variables, f_1,f_2, and f_3. Figure 4 shows alternate ways to accomplish this decomposition. There are many functions which require one more module than their complements to be synthesized. If f_1 or f_2 is of this kind, then the appropriate decomposition should be made to gain this savings. This decision can be postponed until it is known where this savings can be obtained.

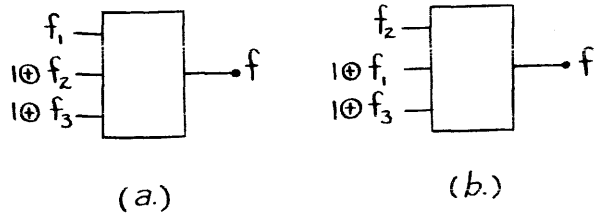


Figure 4—Alternate decompositions of $f = f_1f_2 \oplus f_3$

Often a significant improvement in factoring can occur if a single term is added to the expression. The identity $f_i \oplus f_i = 0$ allows this to be done without changing the truth value of the expression.

Step 2 of the above procedure gives a general guideline of what to look for in a decomposition. At worst, it is always possible to decompose a function of k variables into two functions of k-1 variables by factoring out one of the variables from the terms in which it occurs. For example,

$$f = x_1[f_1(x_2,x_3,\text{—}x_n)] \oplus f_2(x_2,x_3\text{—}x_n)$$

where f is a function of n variables, x_1 is a single variable, and f_1 and f_2 are functions of the remaining n-1 variables. Figure 5 shows the circuit realization for this function.

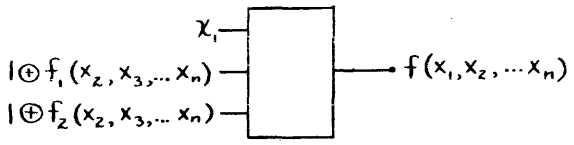


Figure 5—Decomposition of a function of n variables into two functions of n-1 variables

To illustrate this procedure, the following example is offered.

Problem: Construct a circuit of interconnected WOS modules which realizes f = abd+abd+bcdef+abdef+abcdef.

Solution:

(1) The function is first expressed as a ring sum.
f = d ⊕ ad ⊕ bd ⊕ ef ⊕ aef ⊕ bef ⊕ def ⊕ adef ⊕ acef ⊕ bdef ⊕ acdef.

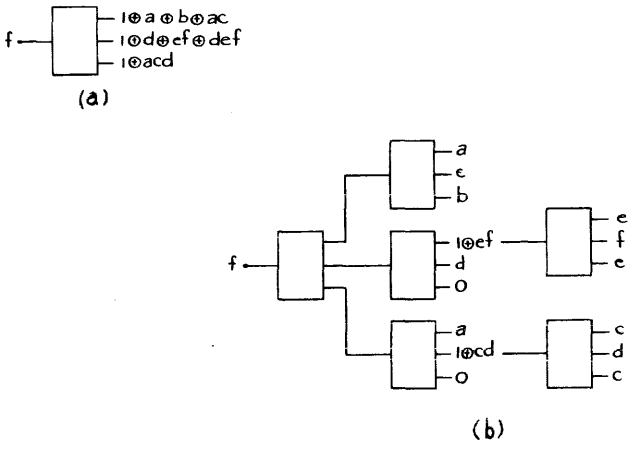


Figure 6—Circuit realization of f = abd + abd + bcdef + abdef + abcdef
(a) After the first decomposition
(b) The complete circuit

(2) If the two terms acd $\oplus$ acd are added, the expression can be factored:
$f = (1 \oplus a \oplus b \oplus ac) \cdot cd \oplus ef \oplus def) \oplus acd$

(3) Since $1 \oplus a \oplus b \oplus ac$ can be realized by a single module, the first decomposition takes the form of figure 6a.

(4) The expression $d \oplus ef \oplus def \oplus 1$ can be factored into $(1 \oplus d)(1 \oplus ef)$. Since the variable d can be obtained without a module, and the expression $1 \oplus ef$ requires one module (the expression ef requires two), the second decomposition sets $f_1 = 1 \oplus ef$ and $f_2 = 1 \oplus d$.

(5) The expression $1 \oplus acd$ has no "best" decomposition. The canonical expansion $a(cd) \oplus 1$ decomposes into $f_1 = a$, $f_2 = cd$, and $f_3 = 1$. The final circuit is shown in Figure 6b.

Summary

The goal of this paper has been to indicate the potential savings which can be achieved with complex building block modules. The results presented, however, have only scratched the surface. Since integrated circuits are now a fact of life, the need for further study can hardly be minimized. Three problem areas are recommended for immediate research:

(1) The establishment of a rigorous set of "goodness" criteria for a building-block module (or set of modules). The arguments presented above for total asymmetry and logical versatility are only intuitive. Perhaps there are other properties (e.g., duality) which affect the value of a building block module.

(2) The development of a minimal synthesis procedure for the WOS module, or perhaps for some other complex module having similar properties. A module is of little value if it cannot be easily used to implement switching functions.

(3) The consideration of the WOS module as the basic cell in a complex array. Current trends in large scale integration consider the unit cell of a large scale array as a single NAND gate. If a more complex cell is used, the resulting array should provide significant savings in the number of cells and complexity of the interconnect pattern needed to synthesize a given function.

ACKNOWLEDGMENT

The author wishes to express his appreciation to certain members of the staffs of the Components Division of the IBM Corporation, East Fishkill, N. Y. and the Sprague Electric Research Laboratory, North Adams, Mass. The author was first introduced to this problem area and many of the initial results were obtained while he was employed as a summer student by IBM. The problems attending the fabrication of the WOS module he has become aware of from his association with Sprague. The author also wishes to acknowledge with thanks the advice and comments of his former thesis advisors at Stanford University, Dr. Richard L. Mattson and Dr. C. Hugh Mays.

REFERENCES

1 Y. N. PATT *Minimal module synthesis of switching functions* Ph.D. Dissertation Stanford University (Available from University Microfilms, Ann Arbor, Michigan) June 1966

2 L. HELLERMAN *A catalogue of three-variable or-invert and and-invert logical circuits* IEEE Trans. on Electronic Computers EC-12 pp. 198-223 June 1963

Adaptive systems of logic networks and binary memories

by I. ALEKSANDER

Queen Mary College, University of London
London, England

INTRODUCTION

Various adaptive circuit schemes have been suggested since the early proposals by Ashby,[1] McCulloch and Pitts,[2] Rosenblatt,[3] and many others. The way in which these schemes were investigated can be divided into three general classes. Firstly, and predominantly, there is simulation by a digital computer; secondly, we find the use of adaptive threshold elements such as Widrow's 'adaline';[4] and finally there are techniques which are realizable by conventional logic and memory circuitry such as Andreae's 'Stella' concept,[5] Vernot's tunnel diode nets,[6] and systems of the 'Artron' type.[7]

This paper suggests a general approach to systems of the third class. These have the advantage of being physically realizable as special purpose digital hardware. Furthermore, neither do they require the awkward analogue memories (variable weights) of threshold devices, nor is it necessary to impose linear separations of the input space[8] in classification tasks. In fact, it will be reasoned that the proposed system can simulate and improve upon the tasks performed by threshold elements.

The circuits in this paper are adaptable by virtue of being able to perform many logic tasks. Such 'multipurpose' devices when treated in the past,[9] were restricted by the need to economise on circuit components. Such limitations need no longer be imposed since the circuits may be fabricated in monolithic form. An awareness of microcircuit techniques is maintained throughout the description of the circuits which follows.

The general adaptive logic system

A general adaptive logic scheme may be pictured as shown in Figure 1. This consists of two salient components: an adaptable logic network and a control memory. The logic network has n binary input channels and m output channels. It is adaptable by virtue

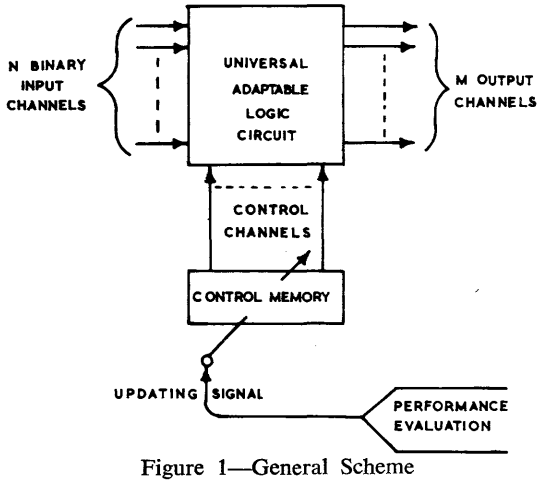


Figure 1—General Scheme

of the fact that the Boolean functions relating the outputs to the inputs may be changed. Here one imposes an important condition: that the logic circuit should be *universally* adaptable. That is, it should be capable of providing *all* the possible combinational relationships between the outputs and the inputs. A particular function is selected by an appropriate message carried by the control wires. These messages are held in the control memory and may, in turn, be updated by external information.

This simple system becomes adaptive once an updating policy has been chosen and its execution is placed under the control of a signal which indicates the performance of the system. Both the updating policy and the nature of the control signal are under the control of the designer.

There are various levels of sophistication at which such a system can be operated. At the simplest level a human operator can change the contents of the memory so as to avoid incorrect actions, whereas in a more elaborate arrangement an automatic evaluator could base the performance signal on the statistical behaviour of the machine, and not only update the

contents of the memory, but also correct a poor updating policy.

We shall return to the operation of various systems later, while at this stage we merely emphasize that the degree of adaptive sophistication does not depend on the universal logic circuits. These are, however, important elements and must be investigated in their own right.

Universal adaptable logic networks

A logic network with n inputs and m outputs is capable of performing an absolute total of

$$2^{m2^n}$$

functions. If the control channels are to select any one of these, they must carry at least this number of messages. In pure binary coding, the least number of control wires is thus $m2^n$.

Here it is proposed that the logic network be separated into m modules, one for each output as shown in Figure 2. Each module is connected to the n inputs and is capable of performing 2^{2^n} functions requiring 2^n of the control wires. This is a convenient separation

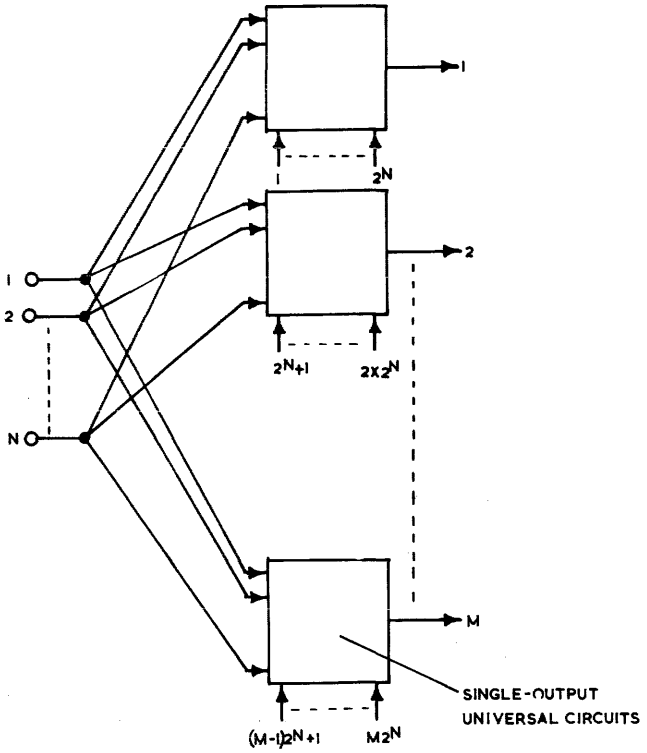


Figure 2—Generation of M outputs

since each output must be capable of being related in each of the 2^{2^n} way to all the inputs, independently of the other outputs. It therefore remains necessary merely to describe the design of an n-input, one-output universal logic network and to realize that an m-output device must contain m of these.

A one-input, one-output universal circuit is the smallest adaptable logic circuit of any use, and is also a building brick with which an n-input device may be synthesized. Let us label its input x_o and its output F. In this case $2^{2^n} = 4$, the four functions being:

$f_o(x_o)$, $F = 0$,
$f_1(x_o)$, $F = x_o$,
$f_2(x_o)$, $F = x_o$
and $f_3(x_o)$, $F = 1$.

This is further illustrated in the truth table (table I) shown below:

Table I—Functions of a single variable, x_o

	F			
x_o	f_o	f_1	f_2	f_3
0	0	0	1	1
1	0	1	0	1

Clearly, two wires are required to convey the four control messages. We label these wires ϕ_o and ϕ_1 and encode them as shown in Table II:

Table II—Code for the control wires

	ϕ_o	ϕ_1
f_o	0	0
f_1	0	1
f_2	1	0
f_3	1	1

Now using Table I as a Karnaugh map with the coding of Table II we derive the complete specification for F:

$$F = \phi_o x_o + \phi_1 x_o \qquad (1)$$

Other control wire codes give forms which are either more complex than (1) or equivalent to it. Examples of circuits which perform function (1) are shown in Figure 3.

Next we consider a two-input element. The complete expression for F may be derived as in the previous case:

$$F = \phi_o x_o x_1 + \phi_1 x_o \bar{x}_1 + \phi_2 \bar{x}_o x_1 + \phi_3 \bar{x}_o \bar{x}_1 \qquad (2)$$

where x_o and x_1 are the two input variables. This expression may now be factored as follows:

$$F = (\phi_o x_1 + \phi_1 \bar{x}_1) x_o + (\phi_2 x_1 + \phi_3 \bar{x}_1) \bar{x}_o \qquad (3)$$

We note that the terms in brackets may be accomplished by one-input universal logic circuits, one for each bracket, and the function completed by the use of an additional one-input circuit as shown in Figure 4. This technique may be extended to n variables in which case the number of one-input universal elements is $(2^n - 1)$. This number of modules is required only if the technology limits the size of a module to that of a one-input universal logic circuit.

Evidently a more advanced technology may enable us to fabricate modules with two or more inputs. A two-input universal module may be made by im-

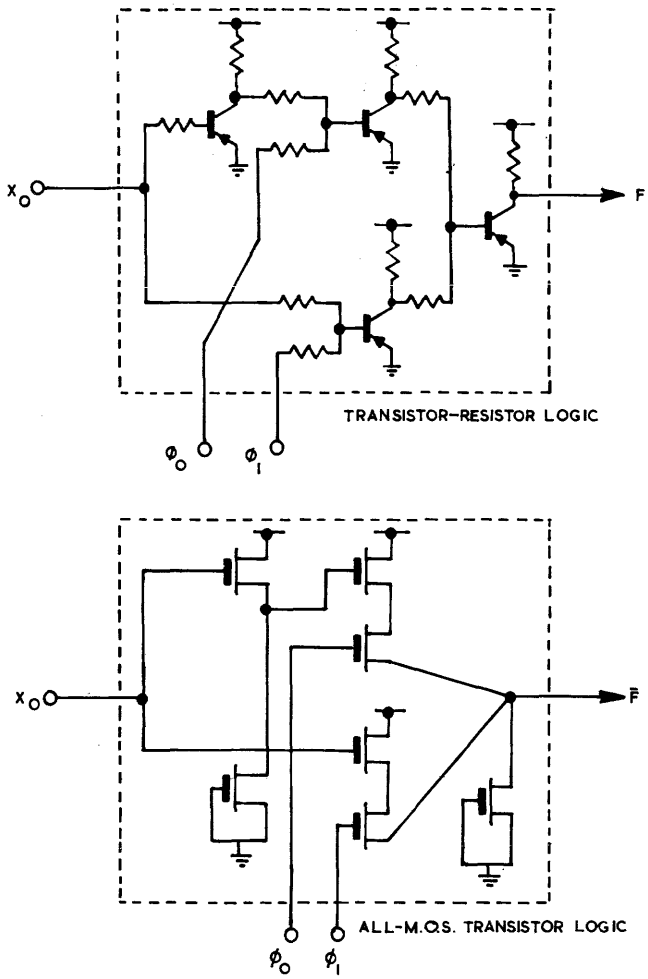


Figure 3—Single-variable universal circuits

plementing equation (2). The general form of this equation for an n-input module[10] is:

$$F = \sum_{j=0}^{2^n-1} \phi_j \prod_{p=0}^{n-1} x_p^{e_p} \qquad (4)$$

or its dual,

$$F = \prod_{j=0}^{2^n-1} \left\{ \phi_j + \left(\sum_{p=0}^{n-1} x_p^{e_p} \right) \right\} \qquad (5)$$

where,

Σ refers to an OR summation,

Π refers to an AND product,

$(e_o, e_1, e_2, \ldots e_{n-1})$ is an n-tuple representing the binary value of j,

and $(\bar{e}_o, \bar{e}_1, \bar{e}_2, \ldots \bar{e}_{n-1})$ is the above n-tuple with the 0's and 1's interchanged.

Also, $x_p^{e_p} = x_p$ if e_p is 0

and $x_p^{e_p} = x_p$ if e_p is 1.

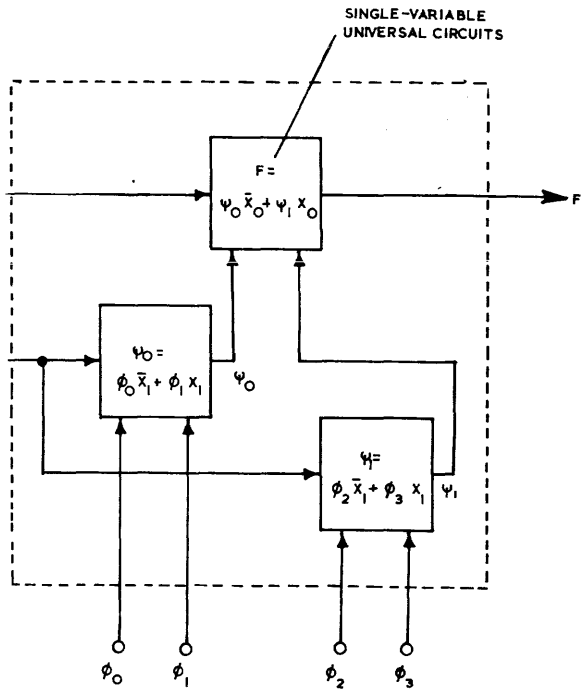


Figure 4—A two-variable universal circuit

The characteristics of such a circuit are given in Table III below.

Table III—Characteristics of an n-input module implementing equation (4) or (5)

Number of gating levels (invertors included)	3
Maximum fan-in	2^n
Maximum fan-out (from invertors)	2^{n-1}
Total number of gates	2^n+n+1
Total number of inputs to gates	$2^n(n+2)+n$

From this table it is possible to select a value of *n* which will yield a feasible module size.

In certain technologies the fan-in of a gate may be limited and this would force an absolute upper bound on *n* for an n-input module. To overcome this, equations (4) and (5) may be factored as (2) was, in order to provide (3). This reduces the fan-in of individual gates, but increases the overall number of gates and the number of switching levels.[10]

A final variation of the above design techniques may be mentioned. Referring to the circuit of equation (4), the fan-in of the AND gates may be reduced, without increasing the number of gates, by increasing the number of wires carrying the input information. For example, the 2^n input messages could be carried by 2^n (instead of n) wires as shown in Figure 5. Here the fan-in of the AND gates is reduced to two. Intermediate steps may be taken by having between *n* and 2^n wires.

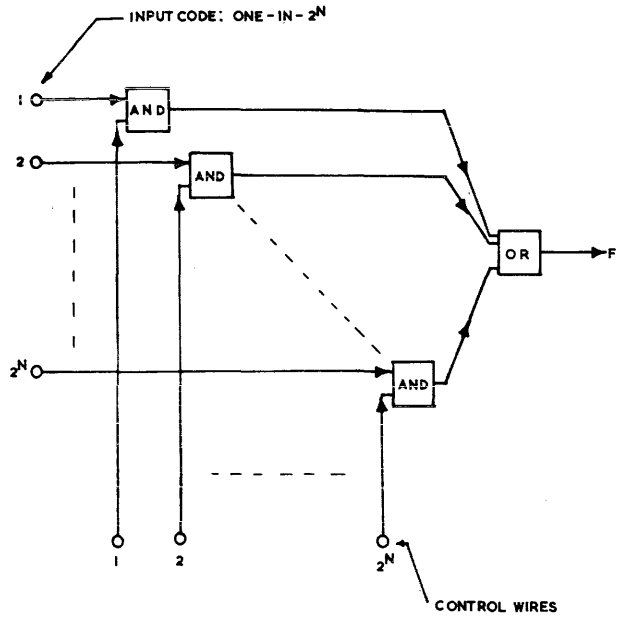


Figure 5—An N-variable scheme with 2^N inputs

The desideratum in microcircuits is, however, to keep the number of lead-in connections as low as possible and therefore a compromise is necessary.

Application examples

In this section some possible applications of universal logic circuits are reviewed and problems relating to the control memory are outlined. The first example is a system which adopts the function of a fixed logic circuit. This scheme is shown in Figure 6. The control memory consists of a code generator which, in the presence of an error signal, generates all the possible binary patterns at the control wires. The generator is stopped and the last-found pattern is held when the error signal disappears. Design details of such a code generator have been discussed elsewhere.[11] This organization of the control memory is limited to circuits with relatively few inputs due to the time taken to search through 2^2 memory messages. The times taken by 10 Mc/s circuits are:

for 1 input : 0.4×10^{-6}secs.,
for 2 inputs : 1.6×10^{-6}secs.,
for 3 inputs : 2.0×10^{-5}secs.,
for 4 inputs : 6.3×10^{-3}secs.,
for 5 inputs : 400 secs = about 3.5 minutes.,
for 6 inputs : 1.6×10^{12} secs = about 50,000 years.

From five to six inputs there is a sharp plunge into unreality.

Thus systems with six or more variables require a more efficient search scheme. For example, the error signal could cause the bits of the memory to change, one by one, until the error disappears. The bit which causes this correction is then locked and not changed if an erroneous response is obtained for another input. This reduces the maximum search time between samples from 2^{2^n} operations to 2^n in the case of a one-output system. Thus the 50,000 years of the previous six-variable example would be reduced to a mere 6.4 microsecs.

Turning now to adaptive schemes capable of some generalization, it is recalled that n-variable threshold devices generalize by virtue of the movement of a hyperplane through the n-dimensional input space. This effect may be simulated in our control memory. As an example, we consider the movement of a separating line through the input space of a two-variable circuit (Figure 7). This is exactly equivalent to the generation

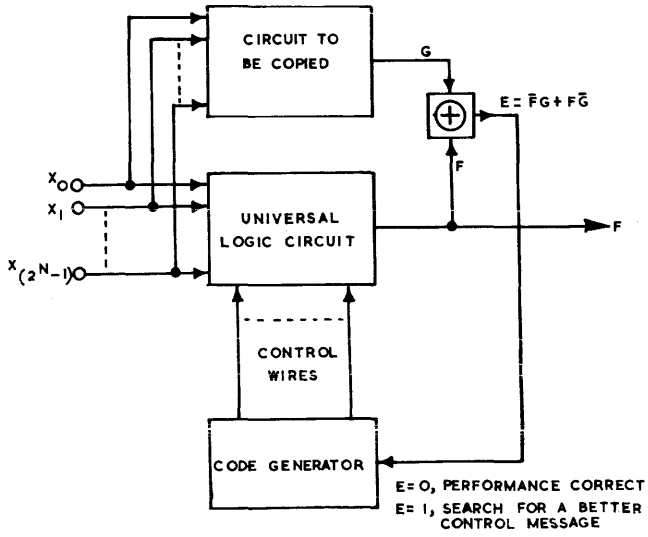


Figure 6—Adaptive circuit copying scheme

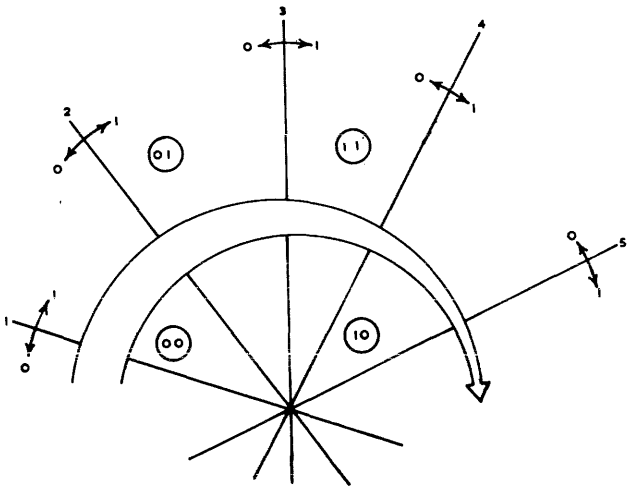


Figure 7—Progression of a linear separation

of the following patterns on the control wires:

	$\phi 0$	$\phi 1$	$\phi 2$	$\phi 3$
position 1:	1	1	1	1
position 2:	0	1	1	1
position 3:	0	0	1	1
position 4:	0	0	1	0
position 5:	0	0	0	0

However, in the logic system there is no need to restrict these sequences to linear separations, and irregular surfaces such as shown in Figure 8 can be made to progress through the input space.

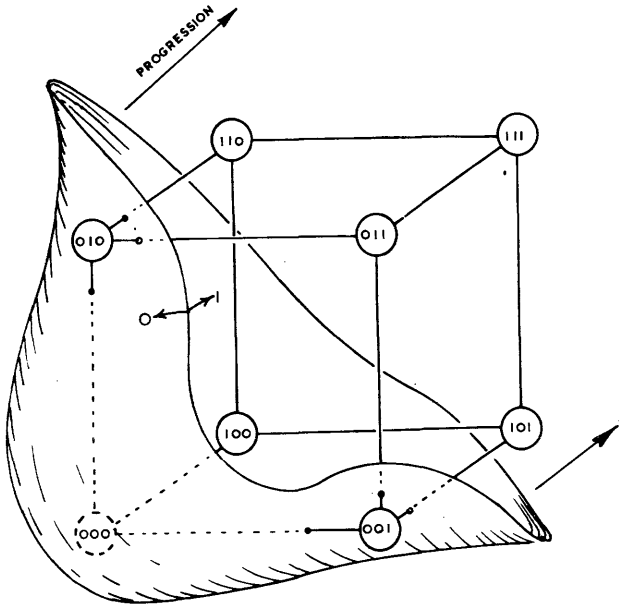


Figure 8—Progression of an irregular surface

So far, only deterministic methods of adaptation have been described. There is no reason for not letting the "learning" procedure be the de-randomizing of initially random memory states in a manner similar to that proposed by Vernot.[6] Another probabilistic technique might use Perceptron-like association units, randomly connected to a sensory field, whose outputs form the inputs of a universal logic circuit and control memory system.

Finally, we speculate that a general purpose processor could be based on the controlled universal logic circuit principle. The data to be processed would be introduced at the inputs of the universal circuits and recovered at their outputs. The control wires would be connected to a large function register, the entire system being backed by conventional data and program storage facilities. Computation would consist of processing the data (numerical or otherwise) through the logic circuits while appropriate control messages were transmitted to the function register. Such a system would be capable of performing any combinational precessing task (e.g., $\sqrt{A}$, LogA etc.) without iteration. Furthermore it could resort to its adaptive modes to perform tasks for which the algorithms were not explicitly specified.

CONCLUSIONS

The subject of logic circuits with controllable universal properties is evidently a vast one. In this paper an attempt has been made to show that this type of logic hardware is the natural development of physically realizable adaptive schemes. The chains tying such systems to models of biological functions have largely been cut and a more powerful system concept has emerged.

The logic design of the data processing section of the system has been detailed since it is easily reasoned that this would be common to equipments operating in various modes and at differing levels of complexity. It appears that the resulting circuits are likely to absorb whatever monolithic technologies may be available and also provide a further challenge to the microcircuit engineer.

A much less precise description has been given of the second salient part of the system: the control memory. It has been stressed, however, that the design policy of such memories determines the power and sophistication of the system. As our examples show, it is the flexibility of this policy which can cause the system to equal and, perhaps, supersede the performance of systems investigated to date. We find this an exciting prospect for future research.

ACKNOWLEDGMENT

The author is most grateful for the helpful discussions he has had with R. C. Albrow, M. Herbert, E. H. Mamdani and R. Shemer, in whose hands lies much of the future development of this subject.

REFERENCES

1 W. ROSS ASHBY
Design of a brain
Chapman and Hall London 1952

2 W. S. McCULLOCH and W. PITTS
A logical calculus of the ideas immiment in nervous activity
Bulletin of Mathematical Biophysics, 5, p. 115 1943

3 F. ROSENBLATT
Principles of neurodynamics: perceptrons and the theory of brain mechanisms
Spartan 1961

4 B. WIDROW and M. E. HOFF
Adaptive switching circuits
Wescon Convention Record, I.R.E. 1961

5 J. H. ANDREAE
Stella: a scheme for a learning machine
Proc. 2nd IFAC Congress Butterworths, 1963

6 R. VERNOT and E. POWERS
A tunnel diode adaptive logic net
Proc. Int. Solid State Circuits Conf. Philadelphia 1962

7 D. R. MOORE and A. C. SPEAKE
New learning machines for future aerospace systems
New Scientist 10 March 1966

8 R. L. MATTSON
A self-organizing binary system
Proc. EJCC p. 212 1959

9 B. DUNHAM and J. NORTH
The use of multipurpose logical devices
Proc. Int. Symp. on the Theory of Switching Harvard U. 1957

10 I. ALEKSANDER
Design of universal logic circuits
Electronics Letters 2, p. 319 August 1966

11 I. ALEKSANDER
Self-adaptive universal logic circuit
Electronics Letters, 2, p. 321 August 1966

Design of diagnosable sequential machines

by ZVI KOHAVI*
Project MAC and Department of Electrical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

PIERRE LAVALLEE*
Applied Research Laboratories
Xerox Corporation
Rochester, New York

INTRODUCTION

A diagnosable sequential machine is one which possesses a distinguishing (or diagnosing) sequence(s) and thus permits us to uniquely identify the various states of the machine by inspecting its response to the distinguishing sequence.

The problem of determining the properties of a synchronous sequential machine by observing its response to various input sequences was introduced by Moore.[1] In his paper Moore further considered the problem of determining whether a given machine accurately describes its terminal behavior as specified by a state table or a state diagram. His approach, however, has very little practical significance since it leads into extremely long experiments. A different procedure for the design of fault detection experiments for sequential machines was introduced by Hennie.[2] This procedure yields good results for machines which possess distinguishing sequences and when the actual circuit has no more states than the correctly operating circuit. For machines which do not have any distinguishing sequences, Hennie's procedure yields very long experiments which makes them impractical. Further development of this approach was done by Kime.[3] From the nature of the problem it seems that the design of fault detection experiments for arbitrary sequential machines will always lead into lengthy experiments which are extremely hard to apply in any practical situation. With the increasing use of modules and integrated circuits it becomes necessary, however, to be able to determine from terminal experiments whether or not a given circuit operates properly. More effort must be made to design circuits which are easy to maintain and to which simple, and practical fault detection experiments can be designed.

The objective of this paper is to present a method for designing sequential circuits in such a way that they will be made to possess special distinguishing sequences and to which there exist very short fault detection experiments. In order to obtain these special and important properties we have to modify the original design and to add additional output logic. Our aim is to determine the minimal amount of additional output logic which is necessary in order to obtain these special properties.

It should be emphasized that the approach presented in this paper yields experiments which are applied only to the terminals of the circuit and not to any point within the circuit. The terminals, however, are predesigned to enable efficient maintenance of the circuit.

The sequential machines considered in this paper are assumed to be finite-state, synchronous, deterministic, strongly connected and completely specified. The machines are of the Mealy[4] model, where the output is a function of both the state and the input.

As an *experiment* on a machine we define the application of input sequences to the input terminals and the recording of the corresponding response from its output terminals. If the experiment is de-

*Formerly, Department of Electrical Engineering, Polytechnic Institute of Brooklyn.

signed to take the machine through all possible transitions in such a way that a definite conclusion can be reached whether or not the machine operates correctly, it is said to be a *fault detection experiment.* At the beginning of an experiment the machine is said to be in the starting state. The experiments discussed in this paper are simple and preset, i.e., it is assumed that only a single copy of the machine is available to the experimenter and that the entire input sequence is predetermined, independently of the outcome of the experiment. An extensive discussion of the various experiments can be found in Gill.[5]

Definition 1

Let M be a sequential machine having n states. An input sequence x_0 is said to be a distinguishing sequence (or a diagnosing sequence) if, when applied to M it yields n different output sequences depending on the initial states. Hence, by observing the response of M to x_0 and if M operates correctly, the initial state of M at the start of x_0 can be determined.

Definition 2

An input sequence y_0 is said to be a homing sequence if the response of M to its application uniquely determines the final state of the machine independently of the starting state. Every reduced sequential machine possesses a homing sequence while only a limited number of machines have distinguishing sequences. Every distinguishing sequence (DS) is also a homing sequence (HS) while the converse is not true.

Let S be the set of states of machine M. An *admissible set* is any subset of S (including S itself) which is known to contain the starting state.

A five state machine M is represented by Table I. It is obvious that M does not have any DS since both states 1 and 2 under 0 input map into state 1 and produce an output of 0, while states 2 and 5 under 1 input map into state 5 with a 1 output.

Table I — Machine M

	N.S.,Z	
P.S.	x = 0	x = 1
1	1,0	4,1
2	1,0	5,1
3	5,0	1,0
4	3,1	4,0
5	2,1	5,1

Our objective is to obtain a machine M′, which contains M, by adding some output logic to M such that M′ will possess any arbitrary predescribed distinguishing sequence, or sequences. In order to arrive, in a systematic manner, to a solution which requires the least amount of additional output terminals the following procedure is proposed.

Definitely diagnosable machines.

The state table of M may be written as illustrated in the top half of Table II. The pair x/z corresponds to input x and output z. The entries of the table are the "next states" corresponding to every input-output pair. For example, from state 1 under input 1 the machine goes to state 4 with an output of 1. This is denoted by entering a 4 in column 1/1 and a dash (−) in column 1/0. In a similar manner the next states of M are entered into the testing table.

The lower half of the testing table consists of all possible admissible pairs and their implications. For example, corresponding to admissible pair 12 is a pair 11 under column 0/0 and a pair 45 under column 1/1. The lower half of the table is derived in a straightforward manner from the top half. The implied entries represent the conditions under which the admissible pairs are distinguishable. Pair 13 is distinguishable by an experiment starting with a 0 if and only if pair 15 is, while pair 12 is indistinguishable by such an experiment since it implies a repeated pair 11. An admissible pair which does not imply any other pair, i.e., all the entries in the corresponding row are dashes (see pair 14), can be omitted from the table (pairs 24 and 35 have therefore been omitted from the testing table). Whenever an entry in the testing table consists of a repeated state (11 in row 12), that entry is circled. A circle around 11 implies that both states 1 and 2 are merged under input 0 into state 1 and hence are indistinguishable by any experiment starting with an input 0. In order to obtain a machine M′ which contains M and possesses a DS which starts with a zero, at first we have to add an output terminal and assign different output states to each state which implies a circled entry.

The testing graph is derived from the testing table. Each node corresponds to an admissible pair. If an admissible pair implies another pair (13 implies 15 in Table II) a directed branch is drawn leading from the admissible pair into the implied pair. Only implied pairs which are not circled (i.e., without repeated states) need to be considered for the derivation of the graph.

The graph for machine M which consists of seven nodes is given in Figure 1. The labelling of the branches corresponds to the top headings of the testing table. An inspection of the graph reveals that it contains a loop 45-23-15-45, i.e., starting from the pair 45 the input sequence 001 does not distinguish

Table II — Testing Table

P. S.	0/0	0/1	1/1	1/0
1	1	—	4	—
2	1	—	5	—
3	5	—	—	1
4	—	3	—	4
5	—	2	5	—
12	11	—	45	—
13	15	—	—	—
14	—	—	—	—
15	—	—	45	—
23	15	—	—	—
25	—	—	55	—
34	—	—	—	14
45	—	23	—	—

between states 4 and 5.

Definition 3

A sequential machine M, having n states, is said to be *definitely diagnosable* (DD) if any sequence of length φ, where $\varphi \leq n(n-1)/2$ is a DS for the admissible set $S = (S_1\ S_2 \ldots S_n)$.

If no repeated states (circled entries) exist in the testing table then there are no two states in M which map into the same next state and yield identical outputs. A loop-free graph guarantees that the longest path includes, at most, $\frac{n}{2}(n-1)$ nodes corresponding to all possible admissible pairs. Hence, the following theorem results.

Theorem 1

The necessary and sufficient conditions for a sequential machine M to be definitely diagnosable is that the corresponding testing graph is loop-free and that no repeated states (circled entries) exist in the testing table.

Theorem 1 defines the conditions under which a sequential machine M is DD. Machine M is therefore not DD since 11 and 55 exist in the testing table and the testing graph is not loop-free. To obtain a corresponding machine M′ such that M′ contains M and is DD we have to augment the output terminals of M. The procedure for modifying the output is summarized as follows:

(i) Eliminate all the circled entries by assigning different output states to the corresponding next state entries.

(ii) Open all the loops of the testing graph by eliminating the smallest number of branches in the graph. A branch is eliminated by assigning different output states to the next state entries which are covered by the node into which it terminates. The choice of branches for elimination is based on two criteria. The first criterion is the reduction of the paths in the graph, i.e., a branch is chosen for elimination if it opens a loop and in addition opens some paths in the graph. The aim is to minimize the length of the longest path in the graph. The second criterion is the minimization of the number of eliminated branches. We shall apply this procedure to machine M.

Step (i) is accomplished by assigning to the outputs associated with the next state entries 1, in column x=0 and states 1 and 2, the values 00 and 01, respectively. Similarly from the circled entry 55 in the testing table we conclude that the outputs associated with the next state entries of states 2 and 5 (column x=1) must be modified to be 10 and 11, respectively. These steps ensure that the testing table of machine M′ is free of repeated entries. Step (ii) is accomplished by opening the loop in the graph. This can be done by eliminating any one of the three branches in the loop. However, the cancellation of the branch from 15 to 45 opens an additional path (13-15-45). After this elimination the longest path includes four nodes (12-45-23-15), this will result in a very long DS and hence, an attempt is made to open this path by the elimination of the branch 45-23. The first branch from 15 to 45 is labeled 1/1, hence states 1 and 5 go, under input 1, into states 4 and 5 respectively with an output of 1. State 5 has a modified output 11 (from the application of rule i), therefore state 1 must be assigned an output of 10. In the same manner the second branch labeled 0/1 from 45 into 32 may be opened by assigning the outputs associated with next states 3 and 2 with distinct outputs 10 and 11, respectivelv.

Table III — Machine M′

P. S.	x = 0	x = 1
1	1,00	4,10
2	1,01	5,10
3	5,00	1,00
4	3,10	4,00
5	2,11	5,11

In the resulting table for machine M′ (Table III), we note that three output entries are incompletely specified. Their specification can be made accord-

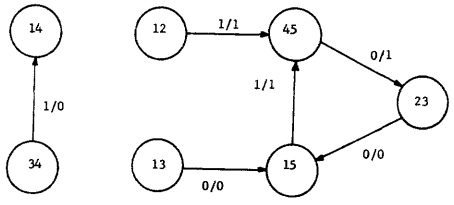


Figure 1 — Testing graph

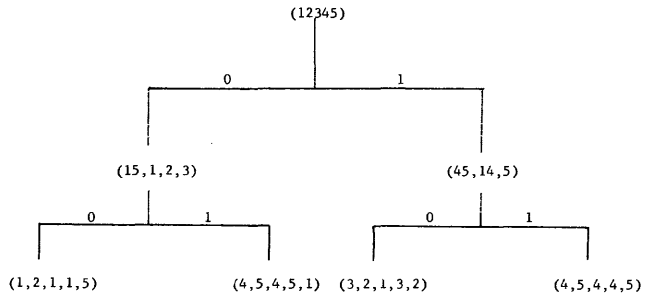


Figure 2 – Diagnosing tree for machine M′

ing to various economic criteria or in order to further open some paths by removing branches from the testing graph and thus shorten the DS. In this particular problem the best we can do is leave these entries unspecified and consider them as don't cares.

The preceding procedure did not affect the next-state or the output behavior of machine M since the output Z of machine M′ is identical to Z of machine M. The augmentation of M results in an addition of the output terminal Z_1. The main advantage of M′ over M is in the fact that M′ is DD and thus efficient checking experiments can be constructed for fault detection in M′ while this is not so easy for machine M.

Theorem 2

Let the longest path in a loop-free testing graph of machine M be m (i.e., the longest path contains m branches) and let the testing table be free of repeated states, then any sequence whose length is, at most, m+1 symbols is a DS on M.

Proof: If the m input labels on the longest path in the testing graph are chosen in their order of appearance as a DS, only the two states (node in the graph) from which this path originates will not be distinguished or yield different responses. The path terminates in a node which corresponds to some pair of states, any other input symbol added to the above sequence will distinguish between these states since no other pair of states is implied.

In machine M′ the longest path in the testing graph (after the elimination of the branches according to steps i and ii) consists of a single branch, hence the DS is of length 2. This is illustrated by the diagnosing tree in Figure 2, from which it is evident that any sequence of length two is a DS on M′.

Theorem 3

To every strongly connected sequential machine M there corresponds another DD sequential machine M′, which is obtained by an addition of some output terminals to M. The original machine may or may not possess any DS.

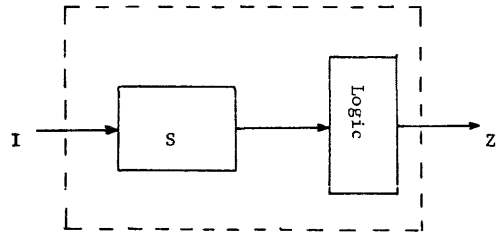
The proof of this theorem is obvious from the preceding discussion and from the construction of M′. For any 2^n-states machine we need, at most, n additional output terminals. However, experience indicates that one or two additional terminals will be sufficient for most machines.

The block diagram of the desired solution is given in Figure 3.

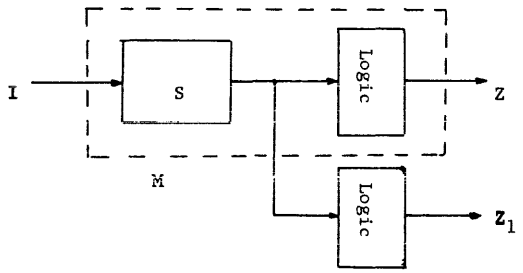
The technique presented in this section is applicable with minor modifications, to the cases of Information Lossless, (IL), Information Lossless of Finite order (ILF) and Finite Memory automata, i.e., to every sequential machine there exists a corresponding machine which is IL and ILF. Similarly to every arbitrary sequential machine there corresponds a finite memory machine. The respective machines are obtained by modifying the output logic of the original ones.

A procedure for the design of checking experiments

The construction of checking experiments for machines which do not possess a DS or simple locating sequences is extremely difficult. Even when some locating sequences can be found the length of the experiment is so great that it becomes almost impractical to apply. The bound on the length of such experiments given by Hennie[2] for the general case is $mn^4(n+1)!$. This is a result which makes the checking experiment impractical to apply repeatedly to any machine. In the previous section we showed that every machine can be modified to become a DD machine. In



(a) Machine M.



Machine M′. Z_1 is used for diagnosing purposes only.

Figure 3 – Illustration of theorem 3

this section we outline a procedure for the construction of checking experiments for DD and a more general class of sequential machines. The experiments are relatively short and it seems that the ease of maintenance and fault detection more than compensates for the minor amount of added logic. We therefore propose to predesign sequential machines with certain DS's by which simple checking experiments can be constructed without the need to obtain any information from the interconnections within the machine.

Let X_o be a DS, of length L, for M. Define $T(S_i, S_j)$ to be a *transfer sequence* which takes M from state S_i into state S_j.

The proposed procedure for the construction of checking experiments involves the use of DS's with repeated symbols, i.e., 000 or 111 etc. For machine M′ choose $X_o = 00$. Assume that M′ is in a starting state 1 at the beginning of the experiment. If it is not in 1 it is always possible to bring it into 1 since every sequence of length 2 is also a HS. In order to simplify the notation we shall use the decimal value of the output, hence 10=2, 11=3 etc.

The experiment starts with the application of the DS $X_o = 00$ to ascertain that M′ is actually in state 1. According to the state table the machine remains in state 1 under 0 input. To verify that 1 is a stable state apply another 0 input following the first X_o. The input to M′, at this stage, consists of three consecutive zeros. The first two zeros serve to check the initial state while the last two zeros (which are the same DS) serve to check a transition from 1 under input 0. The machine remains in state 1 until an input 1 is applied, followed by X_o to check the transition from 1 to 4. Provided that M operates correctly a 1 input takes M to state 4 with an output of 2. X_o takes M into state 5 through state 3. To check the transitions from 4 to 3 and 3 to 5 we keep applying 0 inputs. As long as the machine goes through new transitions we keep applying the same 0 inputs. When a new transition cannot be obtained with an input of 0, a perturbation by means of a 1 input is applied followed by the DS X_o. Assuming that the don't care entries are zeros the checking experiment at this point is as follows:

Input:	0	0	0	1	0	0	0	0	0	0
State:	1	1	1	1	4	3	5	2	1	1
Output:	0	0	0	2	2	0	3	1	0	0

It is evident that an application of another input of 0 does not yield any new transition and hence we must apply an input of 1. Provided we can verify by the end of the experiment that the studied machine contains five states and operates in accordance with Table III, the above experiment serves as a check on the transitions from states 1,2,3,4 and 5 under input of 0 and from state 1 under an input of 1. An application of 1 at this stage of the experiment will not yield any new transition since it takes M into state 4 while the transition from 1 to 4 has already been checked. Hence a transfer sequence is needed to take the machine from state 1 into some state S_j such that the machine goes through "checked" transitions only. This guarantees that the machine actually terminates is S_j if it has operated correctly up to this stage. Apply T(1,4)=1 followed by another 1 and X_o. At the end of this part M is in state 5. Since every transition under 0 input has already been checked a 1 input is applied followed by X_o. This part of the experiment is as follows:

Input:	1	1	0	0	1	0	0
State:	1	4	4	3	5	5	2	1
Output:	2	0	2	0	3	3	1

At this point the only unchecked transitions are from states 3 and 2 under 1 input. Hence, apply T(1,3) = 10 followed by 1 and X_o. This part leaves the machine in state 1. T(1,2) = 1000 is next applied followed by $1X_o$. The complete checking experiment for M′ requires 29 input symbols and is given as follows:

An inspection of the input-output symbols reveals that there are at least 5 states to the machine under examination, since there are five different responses to the application of the input sequence 00, i.e., responses 00, 20, 03, 31, 10. Similarly it is straightforward to show that a machine that satisfies this input-output relations and has five states must be identical to the state table given in Table III (except for relabelling its states). In the same manner we could have constructed an experiment using $X_o = 11$ as a DS.

The upper bound on the length of these experiments is

$$\varphi = n.m + n(m-1).L + L + (m-1) \cdot (n-1)^2$$

where n and m correspond, respectively, to the number of states and distinct inputs in M. This bound is smaller by far than any other known algorithm for the construction of fault detection experiments for sequential machines.

CONCLUSIONS

The testing graph and the associated technique of branch cutting by the addition of output logic has been introduced as a method to embed any sequential ma-

Input:	0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0 0
State	1 1 1 1 4 3 5 2 1 1 1 4 4 3 5 5 2 1 4 3 1 1 1 4 3 5 2 5 2 1
Output	0 0 0 2 2 0 3 1 0 0 2 0 2 0 3 3 1 2 2 0 0 0 2 2 0 3 2 3 1

chine in a definitely diagnosable machine. It can be shown that this technique also applies in embedding a sequential machine which is not IL or ILF into one which has this property; the same applies to the finite memory property. The method answers the question of what is the minimum necessary additional output logic that is needed to obtain a machine with specific properties. This concept is applied and specialized to the problem of checking experiments, where the property that the sequential machine which embeds the original one should have a distinguishing sequence which consists of repeating the same input symbol a certain number of times; this requirement was seen to be a special case of a machine which is definitely diagnosable, and techniques were presented to obtain such properties. It must be emphasized that the compromise that has been adopted in this paper (added logic versus complex and lengthy experiments) is not a very costly one and provides for the first time, a working tool for input-output black box types of experiments as presently the situation exists for integrated circuits. It is felt, however, that more effort is needed in order to better utilize the properties of the DD machines and the numerous distinguishing sequences available. For a longer range study it seems that the properties of the DD machines should be utilized to obtain checking experiments which not only tell us that something is wrong with the machine but ones which indicate what is wrong with it, i.e., some type of error correcting checking experiments.

ACKNOWLEDGMENT

Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

REFERENCES

1 E F MOORE
Gedanken-experiments on sequential machines
Automata Studies C E Shannon and J McCarthy editors
pp 129-153 1956

2 F C HENNIE
Fault detecting experiments for sequential circuits
Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design pp 95-110 1964

3 C R KIME
A failure detection method for sequential circuits
Department of Electrical Engineering University of Iowa
Technical Report # 66-13 January 1966
Also IEEETEC pp 113-115 February 1966

4 G H MEALY
A method for synthesizing sequential circuits
Bell Sys Technical Journal Vol 34 pp 1045-1080
September 1955

5 A GILL
Introduction to the theory of finite-state machines
McGraw Hill Book Company 1962

Large core storage utilization in theory and in practice

by T. A. HUMPHREY
IBM Federal Systems Division
Houston, Texas

INTRODUCTION

Amid the rapid growth of the computer industry and its attendant expansion of capability, it is relatively easy for the industry to overlook items of potentially vast significance. One such item, mass storage, has been used since 1963 for job shop support, as well as for real time support, of manned space exploration. These applications have shown that mass storage can dramatically reduce requirements for main storage and at the same time improve system performance. This paper relates theoretical software implications of mass storage, in particular the LCS, as used to benefit the Manned Spacecraft Center in Houston, Texas. Here, the IBM Federal Systems Division holds a contract with the NASA to provide real time ground support for manned space exploration.

The object of the contract is the RTCC, a Real Time Computer Complex, which is operational 24 hours a day during mission support. There have been two generations of computer systems in the RTCC; the LCS has been used in each. The first generation, based on IBM 7094 Computers, supported the Gemini series and early Apollo series missions. The second generation, just now coming into its maturity, is based on the IBM System/360. The LCS utilization in the first generation RTCC has been previously reported[1] and will be discussed only lightly. Emphasis will be on System/360 applications.

Physical properties of LCS

The LCS is implemented as magnetic core storage. It is electronic as opposed to a mechanical device, and has no moving or rotating parts.

LCS is a product of the IBM System/360 technology, and is identified as the IBM 2361 Large Core Storage.[2] It is packaged into units of one million eight-bit bytes of auxiliary storage with each byte having an associated parity bit. Its cycletime is eight microseconds for each eight bytes (one double word). By interleaving two IBM 2361s, an LCS of two million byte capacity and four-microsecond access time for each sequential double word is created.

Three characteristics distinguished the LCS from other bulk auxiliary storage devices.

- LCS has no latency;
- LCS is addressable at the byte level;
- LCS is available to the CPU for instruction execution.

Absence of latency means that any data or instruction sequence in the LCS can be accessed immediately no matter where the previous access had been made. This provides full freedom in the allocation of bulk storage areas, with no need to consider optimization. Byte addressability permits only the data needed to be retrieved, and thus provides additional flexibility and compactness in allocating bulk storage. The availability for direct accessing by the CPU presents almost unlimited possibilites. The potential of LCS within the computer industry has not been realized fully and the implementation techniques and algorithms for its effective use have been barely considered. However, guidelines and techniques have been developed which use the LCS to a definite advantage.

A simple analysis of main storage requirements

A simple exercise can show that unlimited main storage is not required; but instead, there is some lesser amount required which is application dependent. The key is simple, an application needs only enough main storage to feed its CPU with instructions and data continuously.

Consider a computer that can execute 1,000 instructions per second. Assume that the computer

is given a task which requires five seconds to complete and that the program for this task is a straight line of code, that is it contains no branches and no loops. The instruction storage for this computer in servicing this task alone can now be determined as: (five seconds) × (1,000 instructions/per second) = (5,000 words). The example does not consider the amount of data storage required. Also, most programs are not straight line code. In other words, the ratio of storage required to instructions executed is not generally one to one. This ratio, or *storage index,* does exist and can be determined for any particular application. The storage index, in turn, can be used to estimate the ideal main storage requirements. To do this the time constraint T (defined as the period in which a task must be completed) is used. The ideal main storage requirement necessary to meet this constraint is given by the relationship:

$$\frac{T}{i} \cdot S = \text{main storage requirements}$$

where

T = amount of CPU time required for application
i = instruction rate of the CPU in time/instruction
S = storage index in storage/instruction executed, where storage includes necessary data storage.
where storage includes necessary data storage.

The relationship is most meaningful in a real time or on-line environment where processing tends to be cyclic or has a response limitation. It is also meaningful in a job shop environment, but the expression should be summed and averaged over all applications.

The above is not earth shattering. It says a computer configuration need only have as much main storage as is required to hold the program which it is currently executing. When upgrading a computer configuration where the character of the application is not expected to change, one must upgrade the CPU power and main storage in the same ratio.

The ideal amount of main storage as determined by the (T/i)S relation may not be practical or economical. Main storage allocation algorithms have been developed so the contents of one area of main storage can be feeding the CPU instructions and data, while the contents of another portion of main storage are being exchanged. That is, the instructions and data which will next be used by the CPU are replacing instructions and data that were just used by the CPU. This memory exchanging is accomplished at the cost of CPU time required by the control program to administer the exchange.

Two factors are important when seeking a practical amount of main storage. One is the ratio of the ideal amount of main storage to the practical amount of main storage. As that ratio increases, the administrative cost of memory exchanging increase; hence, the CPU penalty invoked also increases. At some point, the dollar cost of lost CPU capacity equals the dollar savings in reduced main storage. That point establishes a practical amount of main storage.

The second factor is the speed at which main storage can be exchanged, and it is here that LCS enters the picture. As the main storage exchange time increases, so does the amount of main storage required for the exchange process. If the auxiliary storage device has a rotational delay, then memory must be exchanged in large enough segments so that the CPU will have instructions and data available to it during the entire seek and data transmission period. If the LCS is used as the auxiliary storage device, then main storage need only be exchanged in segments large enough to provide instructions and data to the CPU during the transmission interval; thereby eliminating the seek time requirements.

In a theoretical environment, the main storage needed is:

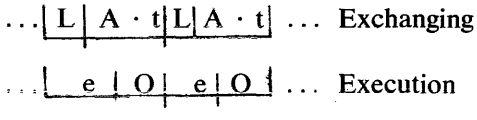
$$\frac{E}{i} \cdot S = \text{bufferable main storage}$$

Here S and i have the same meaning as before, but E is now the average exchange time for a segment instead of the cycle or response time of the system. The exchange time E is dependent on the size of the segments which are exchanged.

In general, the time required to execute a segment should equal or exceed the time required to exchange that segment. This minimizes waiting time in the CPU. This is expressed as:

$$\frac{A \cdot i}{S} + O \geq A \cdot t + L$$

or graphically as:



where:

L = Latency, or access time for storage device in seconds.
A = Exchange segment or allocation

size, in bytes.

t = Transmission rate, in seconds /byte.

e = Execution time of a segment =A(i/S).

O = Exchange overhead of control program, in seconds.

S = Application storage index in bytes/ instruction executed.

i = CPU rate, in seconds/instruction.

The segment size which will balance exchange time with execution time is expressed as:

$$A \geq \frac{L-O}{\frac{i}{S}-t} \qquad \left(\frac{i}{S}-t>O\right)$$

Since main storage must provide space for a segment being exchanged, a segment being executed, and the resident control program, the practical amount of main storage needed in this hypothetical situation is:

2A + control program requirements=total main storage.

The expression for A shows main storage requirements decrease if either latency decreases, or exchange overhead increases. The denominator balances the speed with which the bulk storage device can fill storage (t), against the time required by the CPU to consume storage (i/S). Here the expression shows main storage requirements decrease either as the time to fill main storage decreases or as the time to consume main storage increases. Thus the expression shows that the LCS, with no latency and a high transmission rate, reduces main storage requirements.

Consider LCS, where the latency factor (L) is zero. If main storage can be filled faster than it is consumed by the CPU, then the denominator will remain positive and the minimum exchange size will be a negative quantity. This is a likely case for LCS, and indicates that any segment size can be chosen and the CPU will operate without waiting.

If the denominator goes negative, the original relation is divided by a negative number and the inequality must be reversed. In this case a negative maximum value for A indicates the CPU cannot operate without some waiting.

LCS permit a memory hierarchy

The preceding section was purely hypothetical. It is unlikely that a computer configuration will ever be operated with the practical amount of memory. It is difficult to determine precisely the Exchange Size *A* because S is variable. Even if that could be done, writing programs in just that size would be unenforcable. When considering the LCS, the problem is of more earthly proportions.

The preceding section can, however, be used in a qualitative way. Note the expression which yields the practical amount of memory required. The main difference between the LCS and other auxiliary storage devices is the value L or latency. It is apparent that the LCS does a better job of reducing main storage requirements than does any other auxiliary storage device.

The expression can also be used qualitatively to determine LCS requirements. In reducing main storage requirements, a fantastic requirement for LCS has probably been generated. This is unrealistic, since it is doubtful that Tuesday's payroll application need be in the LCS on Monday.

Not all programs have the same importance or priority. Programs that are not currently in use, or will not soon be in use are not needed. Further, some of the programs which are currently in use may have a shorter response requirement and hence higher priority. Consequently, programs have some hierarchial structure, or priority based on response time required or frequency of use. Frequency of use is important because a program which is used once a second should be retrievable at a lower cost than a program which is used once a day. Frequently used programs are perfect LCS candidates. However, inefficiency in retrieving a program that is used once a day may be satisfactory, and these programs are not LCS candidates.

The LCS does not define hierarchies for programs. It provides the computer community with a tool in which a hierarchy structure can be established and implemented.

There are four ways in which the LCS can, in practice, extend the logical capacity of main storage:

- applications residence,
- systems residence,
- direct CPU access,
- main storage buffer.

When the LCS is used for applications residence or systems residence, it is qualitatively following the expression derived earlier. A small portion of the CPU time is devoted to administering a main storage exchange algorithm. When used as a main storage buffer, the entire system residence and/or application residence does not fit in the LCS. The overflow is being buffered from other auxiliary devices and the LCS is used as intermediate storage. Rather than

suffer the latency penalty at the main storage level, it is paid at the LCS level. Then buffering into main storage from LCS has no latency penalty. When used for direct accesses, the overhead involved in main storage exchanging is avoided, but it is replaced by infrequent degraded memory access time.

System residence on LCS, the simplest step

Using the LCS for system residence is perhaps the easiest way to realize the capabilities of a mass storage device, although no claim is made that such usage would justify the cost of an LCS. System residence on LCS means that components of the system which are not normally resident in main memory will be made resident in an LCS. Each time these components are brought into main storage, a performance improvement over other bulk storage devices will be noted due to the elimination of latency with LCS. The performance improvements are dependent on the job mix at the installation, but the improvements can be dramatic as are indicated in a later section.

The operating system is a logical choice for residence in the LCS because its components are used so frequently in all installations. Additionally, it is the most stable software element in a computer installation, and its components are well known to systems programmers. Because these system components are frequently used their usage is fairly predictable. This makes analysis relatively easy.

Most any scheme for implementing system residence on the LCS will suffice. The benefits are obtained by the speed of the device, not so much by cleverness in implementation. There can be significant decisions which influence flexibility of the implementation, however. A general or a specific implementation can be used, and with either the use of LCS can be reversible or irreversible. By irreversible it is meant that LCS space is permanently allocated to the system components; a sector of LCS is permanently unavailable for normal jobs and processing.

In a general implementation, the LCS is fully supported as an input/output device just as any other bulk storage. System components can then be manually assigned to a storage device of appropriate capability, either a bulk storage device or LCS. Depending upon the operating system, this would be done either at system generation time or whenever the control program is freshly loaded (IPL time).

In a general implementation, the use of LCS tends to be irreversible and may be a drawback. A second disadvantage is the cost of the implementation. A full range of support services must be provided, although only a relative minor group of these may receive heavy use.

A second approach is to specialize the system residence capabilities of the LCS. In this approach, the LCS is not a generally supported device. Rather, the system is programmed to use the LCS in special circumstances. Depending on the circumstances, the process may or may not be reversible. A main advantage for the specific approach is the comparative speed with which system residence on the LCS can be implemented. The main disadvantage is inflexibility. Once the system has been programmed to use the LCS in a certain way, it cannot be changed unless the system is reprogrammed. There would generally be no external handle by which system residence could be altered.

The implementation approach taken for the job shop installation at RTCC is using LCS for system residence was specific and irreversible. In this implementation, residence for SVCLIB and LINKLIB were provided on LCS. (SVCLIB is a library containing the nonresident components of Operating System/360 (OS/360) which provide certain less frequently used application program services, as OPEN and CLOSE. LINKLIB is a library containing certain OS/360 provided routines, as the Assembler, the Fortran compiler, and Link Edit.)

The configuration for which implementation was developed was a System/360 Model 75 with one million bytes of main storage and two million bytes of LCS. OS/360 was modified to reserve, at IPL time for its own use, 667,000 bytes of LCS. OS/360 was also modified to hold additional information in its contents directories. This additional information recorded which system components were in the LCS and their LCS addresses.

OS/360 was further modified to detect the first use of non-resident components following each IPL. Upon each first use, the component was moved from its permanent system residence device to the LCS as well as being brought into main storage. On the second request for that component, it could be fetched from LCS at a greatly reduced access time.

The process of loading the LCS would continue as the system ran until the system residence area was filled. At that time, the system would stabilize. In a minor sense the implementation was job sensitive in that the initial jobs in the job stream would determine which components of system were given LCS residence status.

Application residence benefits operational systems

The LCS merits attention as a residence device for application programs only in, what one might term, an operational system. An operational system is one in which the application program is stable and frequently accessed, thus warranting a specialized residence device. An on-line system is an example of an application which readily falls into this category. Here the application programs are stable and on call 24 hours a day. Any real time system is a candidate, especially one where the response criterion is severe. Certain job shop systems may also fall within the operational category. These could range from a very complex accounting system to a complex scientific program. For a job shop system to merit LCS residence it should be large in terms of the main storage available. It should also be a very significant user of the total loading time used in the installation.

The implementation can be either general or specific. However, under either, the implementation tends to be reversible. In a general implementation, the full range of services to support LCS as an I/O device must be provided. This enables applications programmer at execution time to allocate his programs to LCS externally. In OS/360, this could be done via //DD card (the Operating System/360 Data Definition Statement, used to identify and furnish information about data sets to be used). An obvious advantage is that LCS would be available to the entire installation. A disadvantage is that the implementation would be costly.

A further disadvantage in a general implementation is that effective coordination of the many users is difficult, since responsibility for wise use of the LCS is placed in the hands of each user. While this in many cases is quite satisfactory, the exception may draw severe penalties. First, if the installation has a multijobbing environment, an application programmer will not be able to determine the installation's optimum device allocation for his job because he will not know with whom he is multijobbed. A second philosophical disadvantage applies even in a single jobbing environment if there is an escalation in the size of the application program. Overflowing the available LCS capacity gives the application programmer a constant job of redetermining the optimum device allocation for his programs.

In a specific implementation, the control program is modified to recognize LCS and to administer its use. The way in which it is used can vary in complexity. An advantage of the specific approach is that the benefit of LCS can be realized at a minimum cost in implementation. The disadvantages are inflexibility, and a certain amount of tailoring the control program to an application.

In either a general or specific implementation, the usage of LCS for application residence is generally reversible. That is, LCS would be allocated to applications on a job basis. When the job is complete, the LCS resource would be returned to the system.

Experience is available on LCS application residence in both System/360 based and IBM 7094 Computer based real time systems. The real time control program for the System/360 based system is an extended version of OS/360 called RTOS/360. LCS residence for the application program has been provided by the following modifications to OS/360.

An initializing job step was written, which when executed, prepares the system for real time execution. This job step causes the real time application to be fetched from its home residence on disk and placed into LCS. This fetch is an extension of the fetch which OS/360 executes, in that a special external symbol dictionary is created which identifies the address constants which must be established when the load module is brought into main storage. When the initializing job step is completed, each application load module exists on LCS in a contiguous area and needs only address constants to be relocated into main storage. However, the address constants are resolved by the initializing job step such that the load modules could be executed in LCS.

An LCS directory was implemented to reflect the contents of LCS. When an applications load module is required in real time it is "real time fetched" from LCS. This is a high speed fetch both in terms of no latency on the bulk storage device and in terms of control program overhead required to prepare the load module for execution.

This type of application residence is not basically aimed at scatter loading of load modules. Design control is exercised to keep load modules small so that the main storage supervisor can find space with minimum difficulty. There are exceptions, however, and these load modules are stored on LCS in a scatter fetchable form. The cost of scatter fetching is the extra space in the special external symbol dictionary to hold control section definitions, and the extra main storage supervisor overhead to find space for each individual control section. Scatter fetchable load modules are not scatter fetched unless the main storage supervisor cannot find enough contiguous space in main storage.

Although not discussed to this point, the implementation provided does allow data sets to reside on

LCS. These are specialized data sets, called data tables, adapted for real time use however.

In the 7094 real time system, the only residence device initially recognized during real time was the LCS. When this system was being readied for real time execution, the initializing program transferred all programs and data from the real time system tape to LCS. At the time of the transfer, programs were placed on LCS in a form ready for execution at location zero in main storage. The initialization program was part of the Real Time Executive Control Program (ECP).

A directory was maintained in main storage by the ECP which provided the LCS address of every program and data set. When a required program was not in main storage, it was brought in from the LCS by the memory allocation and IOCS routines. There was special relocate hardware in the 7094 computer which obviated the need for address resolution at this load time. The implementation proved to be very satisfactory, in that it supported the Gemini series missions and the early Apollo missions.

Direct CPU access saves control program overhead

This approach capitalizes on its direct availability to the CPU and causes LCS to be treated exactly as main storage. The use of LCS in this form is really a complicated topic although it appears to be straightforward. The complication lies in the correct use of LCS as a direct substitute for main storage.

By definition, the access time for LCS will be slower than the access time to main storage. Hence, there will be some degradation in CPU performance when accesses are made to LCS rather than to main storage. The amount of degradation will depend upon CPU speed, the LCS access time, the main storage access time, and the frequency of accesses to LCS. Whatever the degradation, it must be compared against the control program overhead to move the program into main storage before it is used. The question is:

$$(\text{degradation/access}) \times (\text{number of accesses/per use}) \overset{?}{=} \frac{(\text{exchange overhead})}{(\text{number of uses/exchange})}$$

The above says that for every CPU access to the LCS there is some degradation. Multiplying this degradation by the number of LCS accesses needed gives the CPU degradation incurred each time the program is executed in LCS. This must be compared to the control program overhead needed to load the program into main storage. Since the program may be used several times per load, the overhead must be divided by the expected number of uses before that program is removed from main storage.

This form of LCS utilization is available both to applications and systems and can either be applied to data or to programs. If applied to data, the likely candidate would be large tables which are accessed randomly. If applied to programs, the candidate would be short but infrequently used routines. A short but infrequently used routine implies that the number of accesses are small compared to the average exchange overhead which would be incurred if brought into main storage.

If this form of LCS utilization is applied to application programs, its use must necessarily fall under control of the application program. This is especially true of using LCS to hold directly accessed data sets, because it would be impossible for the control program to determine the frequency with which an application system accesses its data sets and to make an allocation accordingly. Of course, the comment in an earlier section applies. The individual user may not be in a good position to determine the installation optimum use of LCS.

It would be theoretically possible for a control program to measure the average execution time for application programs and to make a qualitative judgement as to the relative merits of direct execution in LCS or of transferring the load module to main storage for execution. To do this would require some sophisticated timing mechanism and bookkeeping to reflect historical trends of uses of individual programs. This will take considerable overhead in its own right. The state-of-the-art of control programming is some distance from putting these concepts into practice.

Direct CPU accessing of data sets can easily be placed in the hands of users by providing a facility to do a GETMAIN into LCS. This can be implemented by providing an additional argument to the GETMAIN macro. This additional argument signifies that the storage required can be slow access storage. The control program is not obligated to provide slow access storage for this request.

If direct LCS utilization is applied to system programs, there is a higher likihood of immediate success. Here it will be relatively more simple to balance the execution time of certain transient routines against the overhead involved in fetching them into a transient execution area. Routines judged candidates for LCS exectution could then be placed in executable form on LCS at IPL time. The supposition is that this would be an irreversible process. That is the routines selected for direct execution on LCS will always benefit from direct LCS execution, because

they are not subject to fluctuations in their frequency of use according to the job mix at any given time.

The direct use of LCS for tables associated with the control program is not considered worthwwhile. In general the amount of table space required to housekeep and administer a computer are small compared to the main storage available. It would appear relatively immaterial whether these tables are maintained in main storage or in LCS.

LCS buffers main storage

This is perhaps the most interesting, sophisticated, and potentially useful application of LCS. It has been tacitly assumed in discussing LCS utilization for system and application residence, that an LCS of sufficient size was available. This may not be the case. There may be more components to place on LCS than available LCS space. This situation does not necessarily indicate a need to purchase additional LCS or to sacrifice performance improvement, by now buffering from some slower auxiliary device directly into main storage.

There is a considerable penalty for buffering programs and data directly into main storage from a slow bulk storage. Main storage must be allocated and reserved from the moment the request is initiated. This main storage must be held and is not available for useful execution while the bulk storage device is being accessed. To have sufficient main storage for normal needs, additional main storage can be provided in LCS at a greatly reduced dollar cost. Once programs are buffered into LCS, they can be transferred to main storage for execution at a minimum cost in main storage buffer areas.

To consider what to do when the LCS is full, consider its purpose. In the case of system and application residence the LCS is a reservoir for the most frequently required programs and data so that they may be brought into main storage for execution in a minimal time. Conceptually, it is simple to keep in LCS those programs and data which are most likely to be needed in main storage. The implementation is not simple however.

Main storage managment is well defined today. However, LCS management is not so well defined, principally because of the fewer "handles" available to evaluate potential need of its contents. Main storage management deals with immediate needs. The control program senses an immediate need for a program and goes about bringing it into main storage. It has a direct knowledge of what is required. Some light housekeeping can be done on recent frequency of use to assist in determining what can best be overwritten when main storage needs to be exchanged.

LCS management deals with expected needs or potential needs rather than immediate needs and has no good forecast of potential need. One way to solve this problem is to have the user participate in LCS management. This was done on the 7094 based RTCC by identifying logical units of the application system and associated with each the programs and data required by it. Each unit or activity was given an identity and the items required by the unit were placed in an activity list. The application program was then requested to activate the activity prior to its use. This enabled the control program to allocate LCS using an algorithm very much akin to that for finding space in main storage.

The 7094 implementation was good in its single jobbing environment. However, it would seem wise in a multijobbing environment to minimize the need for application users to determine the best allocation of computer resources. The dilemma is compounded by the relatively long time required to exchange or modify the contents of an LCS from a disk, tape or other auxiliary device. Since the exchange time may be in terms of minutes, main storage management housekeeping values such as usage count have little value for forecasting the needs for programs and data three, four, or five minutes ahead.

Of the choices available, an LCS management scheme based on user participation seems the best at the moment.

Performance improvements with LCS

Performance improvements due to LCS utilization on job shop operations on the 7094 under IBSYS and on System/360 under OS/360 are given in Table I. They are indicative only. No attempt to extrapolate these should be made. Each installation must be considered individually. These are observed direct comparisons.

Utilization of LCS on the 7094 for job shop under IBSYS yielded a 30 percent overall decrease in the average IBJOB execution time. These savings were attributable to placing MAP, FTC, IOCS, and IBJOB on the LCS at the beginning of execution of a job stack. Accesses to the system tape were virtually eliminated during multiple compilation or jobs requiring no execution. Not all accesses to the system tape could be eliminated because LCS was available to application programmers and was used in the real time system. If destroyed by a job, the components

Table I
Indicative job shop performance improvements with LCS

Environment	Performance
Installation Job Shop on 7094 under IBSYS; MAP, FTC, IOCS, IBJOB, sometimes on LCS	30% decrease in average IBJOB execution time
Job stack timing on System/360 Mod 75; SVCLIB and LINK-LIB on LCS	Stack ran in 19 min. 21 secs. with LCS residence compared to 49 min. 9 secs. without
Installation job shop on System/360 under RTOS/360. SVCLIB and LINKLIB on LCS	65% decrease in average job execution time with LCS

were accessed from the system tape during that job but were restored onto LCS before the next job.

The performance improvements for System/360 are more impressive than those for IBSYS. When LCS was utilized as system residence on the Model 75 there was a 60 percent decrease in the execution time of a sample job stack. The particular job stack in question required 49 minutes 9 seconds to execute prior to establishing system residence on LCS. After the SVCLIB and LINKLIB were placed on the LCS, this job stack executed in 19 minutes 21 seconds.

Another System/360 comparison is available from the utilization reports of job shop operations. Average job execution time, as obtained by dividing the number of jobs run into the total time including maintenance, etc., was 0.7 hours before using LCS and was 0.25 nours after using LCS.

Performance improvements can be cited for real time operations. However, the operation of the RTCC without LCS is impossible and direct comparisions are not available. Modeling studies of the real time system are used to see what performance would have been without LCS.

It is difficult to cite an adequate performance measurement for LCS utilization in support of real time. In critical phases of flight where response is of paramount importance, the system simply does not work without the immediate access provided by LCS. Perhaps the simplest dramatization comes from the 7094 RTCC and was given before.[1] Table II indicates the useful processing accomplished as a function of bulk storage device hypothetically used.

The figures in Table II indicate how much of the CPU can be used if main storage is being supplied by three different bulk storage devices. The balance of the CPU time is needed in waiting to access programs and data. These figures are approximate and are based on detailed model studies.

Table II

Indicative real time system improvement with LCS

DEVICE PROVIDING AUXILIARY STORAGE

USABLE 65K 70944II CPU CAPACITY
IN A CERTAIN LARGE, HIGH
RESPONSE REAL TIME ENVIRONMENT

LCS (2361-2)	DRUM (7320)	DISK (2311)
75%	20%	7%

LCS configuration analysis

LCS is an expensive piece of computer equipment and requires careful analysis to assure it is being utilized in the best possible way. Experience in developing the RTCC has shown that the most effective way to analyze possible LCS uses is through simulation modeling of the computer system and environment which is a candidate for LCS.

To date, two basic tools have evolved to assist in this type of analysis. They have not been developed specifically for analyzing LCS, but their capabilities extend to it. These tools are models of the control program and System/360 hardware written in the GPSS language.

One model, GMOS, is at the load module and control program service–subservice level. The time scale is in microseconds. The model is used for a detailed analysis of one application.

The second model, SMS, is at the job step and control program service level. Its time scale is in hundredths of a second. It is used for a less detailed analysis of an application or for analysis of a job stream.

The models are configurable to any hardware system. By configuring LCS in and out of a total system model, one can see the predicted performance improvement caused by LCS. A system model is developed by combining one of these models with a model of the application or application job stream. These techniques permit solidly based cost analyses to be made. Figure 1 illustrates the type of information which can be produced by these techniques and the techniques of relating main storage size and bulk storage device to system performance. The X axis

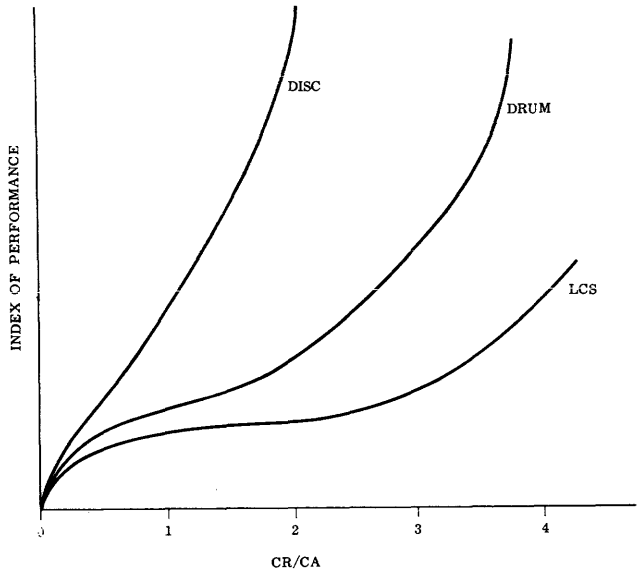


Figure 1 – Typical correlation of main storage size, auxilliary storage type, and system performance

is labeled CR/CA, or the Core Required to Core Available ratio.

CR/CA is a measure of the size of the main storage as a function of the size of the application program. At a ratio of 1.0 the application is an in core system. At a ratio of 4.0 main storage is large enough to contain only 25 percent of the application system. The Y axis, Index of Performance, would depend on the application being studied. In a real time system the index of performance would likely be response. In a job shop environment, the index of performance would likely be average job execution time.

In a job shop environment, the index of performance could also be CPU waiting time. Waiting time may be more significant than average job execution time. An attractive average execution time may be being provided at a high dollar cost for the CPU. If waiting time is high, a slower CPU may give substantially the same performance.

In any event, this type of analysis presents configuration alternatives in a manner readily available for cost analysis.

A further word about the GMOS and SMS models. They have been produced through the expenditure of some five to six man years of effort. They represent the hardware and control program at a very detailed level. They have been calibrated at certain points by comparing their predicted system performance against actual observed system performance and have been found quite reliable. The point to be made is that, while this type of analysis requires models of this detail and accuracy, one must not underestimate the magnitude of the effort involved.

LCS, we are just learning about it

The safest thing that can be said about LCS is that its use is not fully defined. In one real time application it made the difference between success and failure. When directed to job shop applications, its utilization reduced the average job execution time from 30 to 60 per cent.

These results, especially in the job shop environment, have been achieved with relatively unsophisticated techniques. The full potential of LCS as an extension for main storage has not been explored or exploited.

ACKNOWLEDGMENT

The author gratefully acknowledges the constructive critiques of this paper given by H. F. Hertel, J. L. Johnstone, B. F. Minard, W. I. Stanley, and P. W. Weiler.

REFERENCES

1 D F JENKINS J H MUELLER
A strategy for using LCS
SHARE XXV Meeting 1965

2 IBM Publication SRL-28-822-6869-1
2361 Core Storage Original Equipment manufacturers information

Extended core storage for the control data 64 6600 systems

by GALE A. JALLEN
Control Data Corporation
Minneapolis, Minnesota

INTRODUCTION

The software operating systems for the Control Data 64/6600 Extended Core Storage (ECS) have been described in another paper.[1] The ESC is organized basically as a very large word two wire magnetic core memory with multi-phased banks. The use of multi-bank phasing and very large memory word construction has allowed the basic core to operate at a reasonable cycle time of 3.2 microseconds while providing an average data rate of 600 mega bits per second. The purpose of this paper is to describe the design of the basic memory hardware required to operate a very large word memory.

Hardware system description

The memory system is organized in logically independent banks of 125 K words. These banks are made up of 16,384 memory words of 488 bits each. The 488 bit word consists of eight 60 bit computer words plus 1 bit of parity for each computer word. It is a conventional two wire word organized magnetic core system with a cycle time of 3.2 microseconds for each of its 488 bit memory words. The access time to this word is approximately 1.6 microseconds measured at the controller interface. This bank represents the smallest available size for ECS and also the slowest data rate of 150 mega bits per second. Four of these banks can be phased to provide a capacity of 500 K words of 60 bits each and a maximum data rate of 600 mega bits per second. The four banks are housed in one cabinet similar to that of a 6600. Four of these cabinets or bays can then be interfaced with the central processor by means of the controller channels to provide up to 2×10^6 words of core memory.

Figure 1 shows the basic data flow diagram for 1 bank of ECS. Beginning with the CPU coupler in the central processor, the responsibilities assigned to it are: to generate and assemble the ECS address and word count, to update the ECS address for each eight word record transfer, and to provide any necessary ECS and central memory data transfer housekeeping. The ECS coupler communicates with the controller on a 60 bit bidirectional data bus and an independent 24 bit address bus. The controller contains several function responsibilities. It contains a sequential scanning mechansim to service four 60 bit bidirectional data channels from CPU couplers and also four 60 bit bidirectional channels to ECS core store I/O registers. The requests are honored in sequence at the end of each eight word record transferred. Only one data path through the coupler is provided. This data path contains a parity generator with which parity is generated and checked as data is passed through the controller. The status of this parity is transmitted back to the CPU coupler. Since the ECS

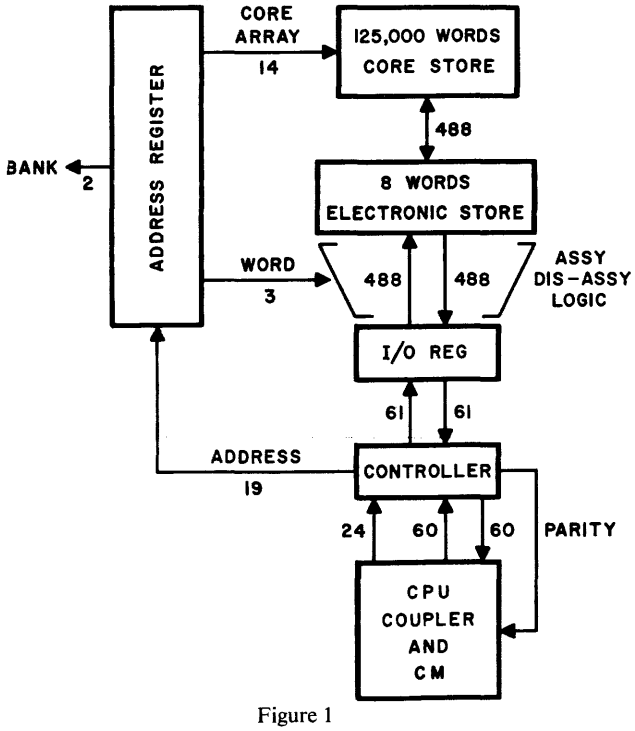


Figure 1

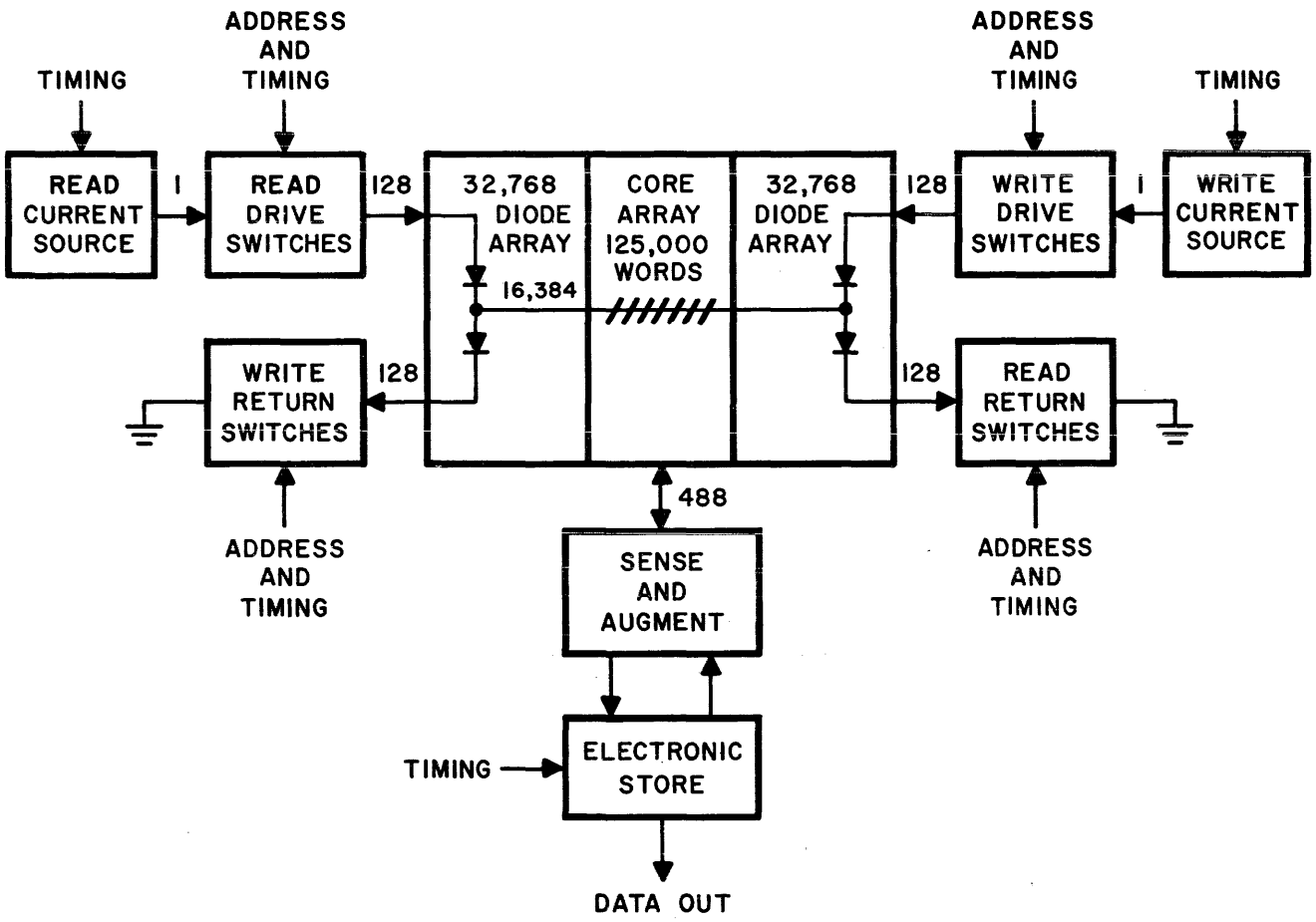


Figure 2

is a synchronized memory system, the controller contains the system master clock which controls all ECS bays and all CPU devices connected into the system. A portion of the 24 bit address information which is sent from the CPU coupler is decoded at the controller to select the bay address. A bay of ECS is 500 K words. The controller also contains ECS core housekeeping logic which provides the necessary timing to insure that no shorter period than 3.2 microseconds occurs between requests to any ECS bank. Moving up through the signal flow on Figure 1, we now communicate with the ECS I/O register. This 60 bit I/O register is shared by the four ECS banks in each bay. The I/O register communicates with the eight words of electronic store through assembly and disassembly logic. This logic is addressed by 3 bits of the ECS address and allows the selection of any part or all of the eight words of the electronic store. The eight words of the electronic store are duplicated for each bank of core memory. The electronic store provides a buffer for holding the data while another bank is being cycled 800 nanoseconds later in a multi-bank phased system. The electronic store also provides the necessary buffer for the recirculation of the data back into store. If

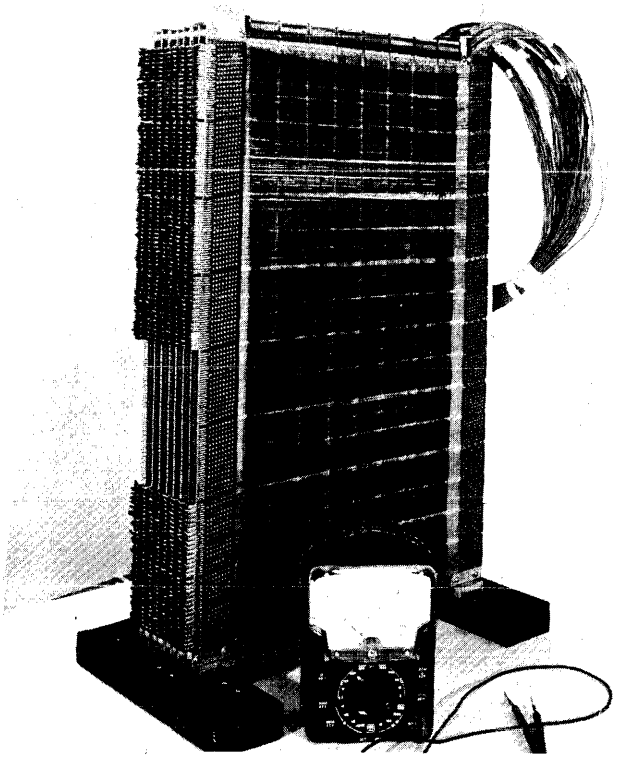


Figure 3

less then eight words are addressed in a write mode, this register recirculates the remaining words back to core on either a read or write operation. The eight words of the electronic store communicate through the sense amplifier augment generator system of the core store which in turn communicate directly with the magnetic core elements. It is this portion of the memory which I will describe in more detail.

Hardware description

Referring to Figure 2 shows the basic block diagram of the ECS core memory proper. The heart of this block diagram is of course the central core array consisting of approximately 8×10^6 cores and their associated selection diode arrays. These items are combined in one core stack system. A picture of this core stack can be seen by referring to Figure 3 which includes a standard voltmeter for size comparison. It is basically 30 inches by 20 inches by 6 inches in size. The basic storage element used is a 30 mil core with a total switching period of approximately 500 nanoseconds. These cores are then mounted on both surfaces of a plane of approximately ½ million cores per side. Also integral with this plane are the word selection diodes.

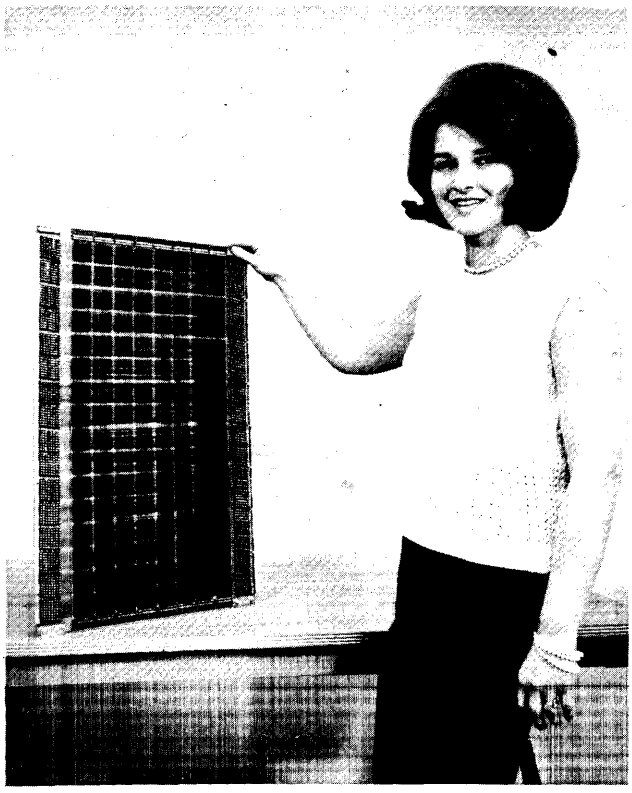


Figure 4

Figure 4 shows a picture of a core plane. Visible in the photo are approximately 5×10^5 cores. They are mounted on a copper ground plane. Along the long dimensions are the selection diodes which form the terminations for the word wires. Along one long dimension edge and both short dimensions are the interconnecting connectors. These connectors are used during plane testing as well as semipermanent stack assembly connections. The back side of the plane consists of an identical set of cores and diodes resulting in a plane assembly of 1×10^6 cores. These plane assemblies are complete units which can be tested and repaired independently before being assembled into stacks. The word selection diodes were placed on the plane assemblies in order to reduce the total number of interconnecting wires required to assemble these planes into a stack of approximately 8×10^6 cores. These diodes are the only electronic assemblies mounted in this area. Approximately 100 watts of power is dissipated in this stack during a write operation. This heat is removed by a low velocity air stream pulled through the stack by small fans located on the edge of the stack which draw cool air from the chilled logic card area which is Freon cooled in the 6000 systems. The assembly of eight planes of 1×10^6 cores each into a stack results in a total bidirectional word drive matrix of 128 by 128. No attempt was made to break this large diode matrix down into smaller portions. Electronic means were used to reduce the effects of the very large capacitance resulting in this array.

Referring to Figure 5, a simplified schematic of the word drive matrix is shown with some of the stray capacitances placed at their physical locations. The primary path for current flow during the read portion of the cycle is shown in bold. A generator labled back emf is shown in series with the selected line and represents the total output of all cores being switched from "1" to "0" state. Since each core produces approximately 50 millivolts, the maximum back emf reaches 25 volts when all 488 bits are in the "1" state. Added to this voltage are diode and IR drops which bring the total to 30 volts. Conversely when all bits are in a "0" state, the total drop reduces to approximately 7.5 volts. This variation in load on the constant current source requires that it have a high output impedance in order that the "bite"[2] into the read current will not be large enough to increase the switching time of the cores as "1's" are added to the word. The typical "bite" experiences on ECS is in the order of 5% when the core switching time totals 350 nanoseconds. This represents a total drive system shunt impedance of approximately 1 K ohm at the core switching frequency of approximately 2.0 megacycles. Referring again to Figure 5, it can be shown that a parasitic capacitance of some 3800 pico farads exists in the drive matrix which is potentially in shunt with the read current source. At 2.0 megacycles this would

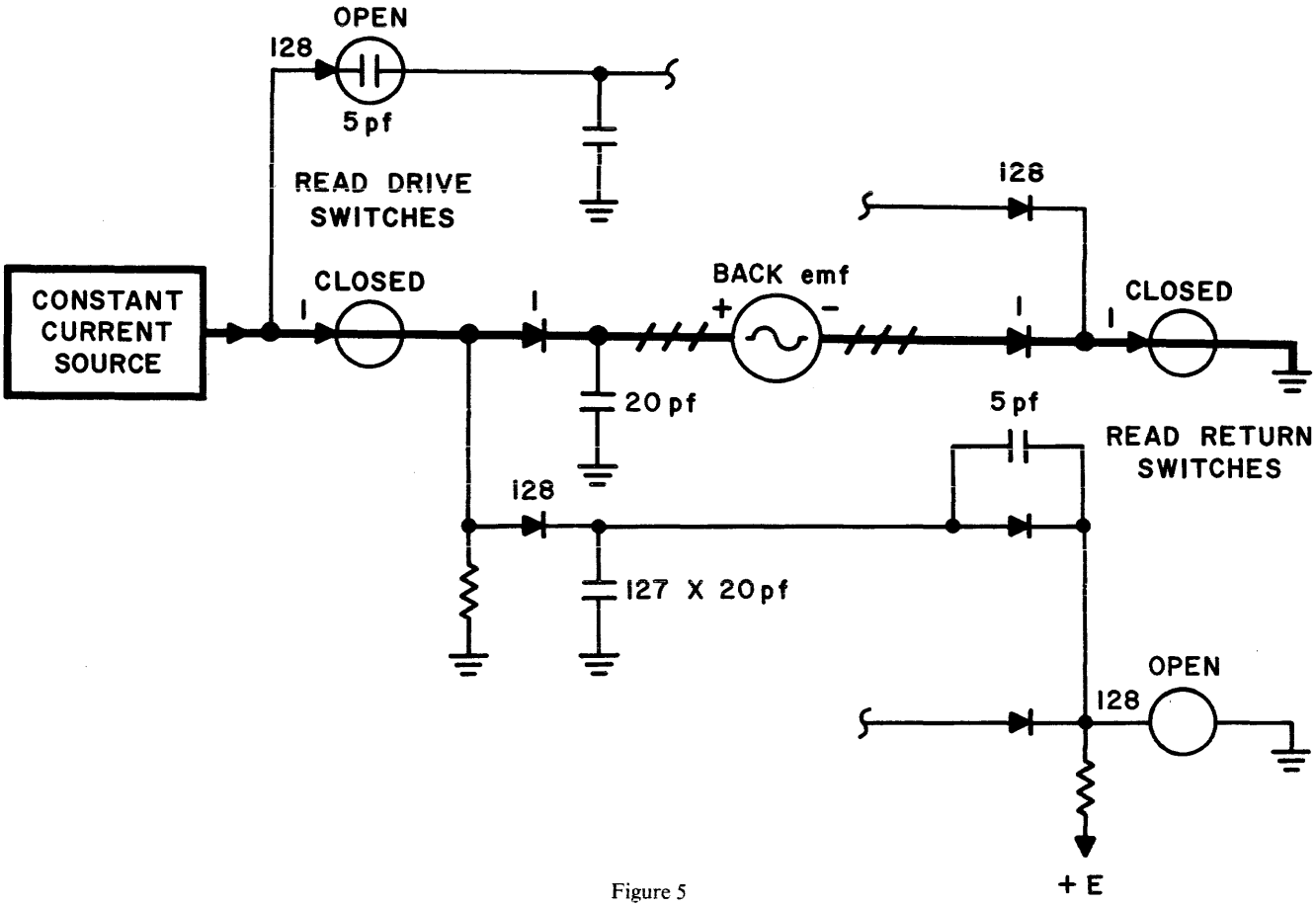


Figure 5

be approximately 21 ohms of shunt reactance, an impossible situation. A system of preconditioning is used in the drive system to reduce by approximately fifty to one the effect of this shunt capacitance. Since some form of damping is required on any LC network, and the condition for critical damping is $R = \frac{1}{2}\sqrt{\frac{L}{C}}$ it becomes apparent that reduction of the effective capacitance allows a higher damping resistance to be used to further increase the effective shunt impedance in the drive system.

Only one read current source is used by the entire ECS bank. The timing and rise time characteristics of this one source determine the entire core array timing. This current source is steered through transformer coupled drive switches. Since these switches are A.C. coupled, any failure of the timing mechanism cannot sustain current in the core array and cause serious damage. As can be seen by referring to Figure 2, two sets of switches are required to provide a path for the current through this core array. All the switches used can be divided into two categories, the read drive switches and the return switches. They are identical in characteristics except for the common terminal, in one case the positive collector is common and in the other case the negative emitter is common in the switch. The write current drive is identical to the read current except for timing and magnitude. In general, the word drive system can be considered quite conventional in the state-of-the-art except for the burden of the large array capacitance and large back emf resulting from long core words.

Referencing Figure 6 shows a simplified schematic of the sense digit system showing the folded sense line. This line totals approximately 40 feet of wire and has 16,384 cores on it. One-half of the 16,384 cores are on each half of this line and are sensed at the end by means of a differential sense amplifier which has A.C. coupling to provide low frequency roll off. Since the sense line is approximately 40 feet long, a rather long propagation delay results and a somewhat peculiar problem of core output summing in the differential amplifier occurs. Cores located at one end of the digit line would appear reduced in output since the positive or negative input to the differential amplifier would be displaced in time before being added in the amplifier. This reduction in output would not occur on cores located near the center of the digit line since both inputs would be delayed an equal amount. This variation in output can be controlled by specifying a core switching time that is long with respect to the digit line propagation delay.

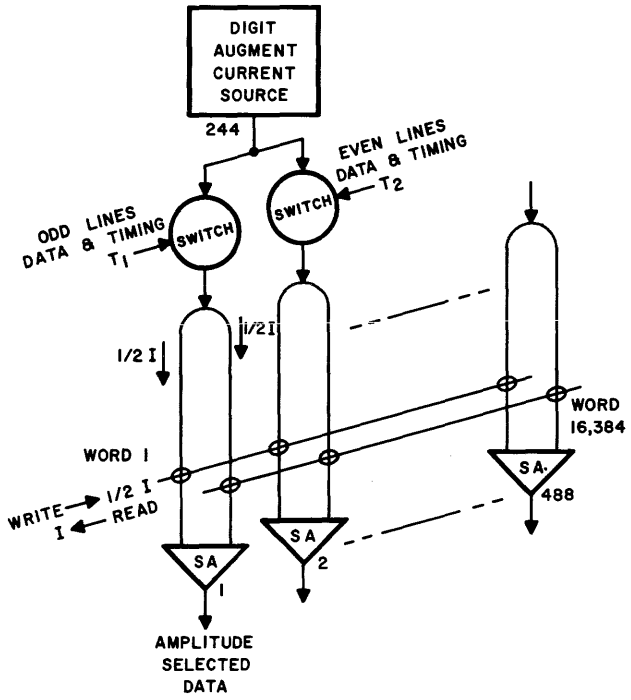


Figure 6

The variation experienced on ECS with a 200 nanosecond propagation delay and a core switching time of 500 nanoseconds is less than 30%.

In order to reduce the length of all lines involved in the core stack, the 30 mil cores were placed on 25 mil centers in both coordinates. This close spacing results in a large mutual capacitance and inductance between adjacent digit wires.[3] A large signal crosstalk and a data dependent digit line characteristic impedance are two deleterious effects of these high mutuals. To reduce the effect of these mutuals, the core mats were placed on a substantial ground system and a sequential odd-even digit line drive was incorporated. Sequencing the digit line drive reduced the mutual capacitances and inductance by doubling the effective digit line spacing. As can be seen in Figure 6, this sequencing also allowed the sharing of digit current sources. This sharing reduces the digit D.C. power 50% by reducing the peak digit current required. The simultaneous drive of 244 digit line pairs results in approximately 120 amperes of current flow through the digit lines and stack ground system during a write operation. In order that recovery from this large transient be effected quickly, all effort was made to maintain a stable characteristic impedance in the digit system of ECS. The results of this effort produced a recovery time of less than 1 microsecond.

Mechanical construction

Referring to Figure 7, a picture of the ECS chassis can be seen. For those familiar with the 6000 series computer construction,[4] the ECS chassis is identical in that a Freon cooling system is used and all logic is constructed on cordwood packages using silicon logic. Approximately 3 ton of refrigeration capacity is required to cool four ECS banks. The waste heat from these systems is carried away by chilled water. Except for the 60 cycle Freon compressor, all power for ECS operation is obtained from a 400 cycle power system. The logic power supply is mounted integral with each chassis. The digit drive power is located remote in the cable exchange area at the end of the ECS cabinet. Four of these power supplies are located in this area, one for each ECS bank.

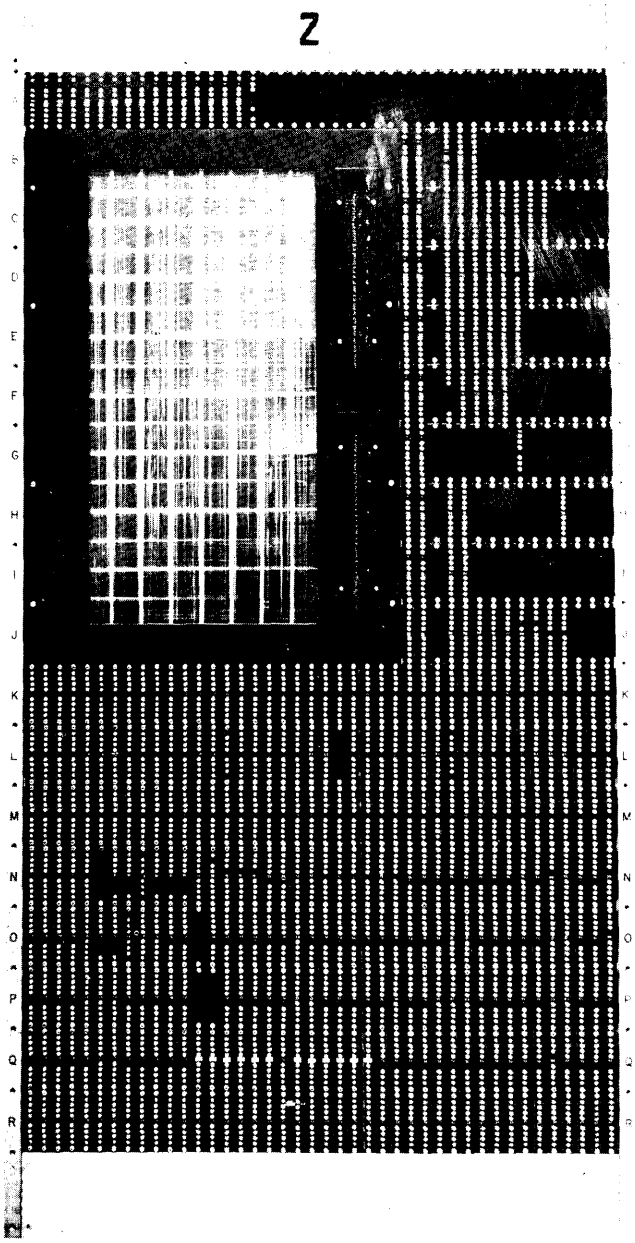


Figure 7

Maintenance

Several maintenance features are included in ECS. All power supplies are made adjustable at the maintenance panel. The purpose of this feature is to help detect potentially marginal situations during scheduled maintenance periods.

A procedure for "graceful degradation" is available. The purpose of this feature is to keep the addressing continuous by pushing down higher order addresses to replace terminated lower order addresses. The effect upon the user is to maintain a continuous address field by accumulating all terminated addresses at the higher order end of the address field. Implementation of this feature requires a minimum of down time and allows the accumulation of several failures before major repair is scheduled.

CONCLUSION

Block transfer or streaming data flow has effected a new approach to memory design. The 64/6600 ECS has taken advantage of this streaming data flow to improve the efficiency of the memory by providing very high capacity and data rates at low cost per bit. Although the ECS was optimized for block transfers, good performance in a random mode is also possible.

REFERENCES

1 M H MacDOUGALL
Simulation of an ECS Based Operating System

2 ASTM Standard C526 63T Test Method for Coincident Current Memory Cores

3 E F TERMAN
Radio Engineers Handbook
McGraw Hill Book Company, Inc
New York 1943 1st ed p 47 109

4 J E THORNTON
Parallel Operation in the Control Data 6600
AFIPS Conference Proceedings Vol 26 Part II p 33 1964

Simulation of an ECS-based operating system

by M. H. MacDOUGALL
Control Data Corporation
Palo Alto, California

INTRODUCTION

Extended Core Storage (ECS) for the Control Data 64/6600 System has been described in another paper.[1] ECS is a large capacity, word-addressable core memory with block transfer times of from eight to ten words per microsecond after a transfer start-up time of approximately 2 microseconds. ECS memories range in size from 131K to over two million words, and can be shared by up to four 6000 systems. The cost per word of Extended Core Storage is about one-tenth that of Central Memory (CM).

The availability of second-level storage with instantaneous access and very high transfer time offers some exciting possibilities for the design of both multiprogrammed and multi-access systems, particularly since the combination of speed and capacity offers simplicity of design as well as flexibility. In order to evaluate design alternatives for an ECS-based operating system, an operating system simulator has been employed. The purpose of this paper is to describe the design of the simulator and to present some of the results obtained via simulation. First, let us review the basic elements of a proposed Extended Core operating system (ECOS) and the environment in which it resides.

System description

The basic environment for ECOS is illustrated in figure 1. The 6000 system[2] has twelve data channels, to which a variety of peripheral devices may be connected. There are ten peripheral processors, each with its own one-microsecond memory of 4096 12-bit words. A peripheral processor (PPU) can connect itself to any one of the twelve data channels and initiate an input or output operation; it can transfer a block of words to or from central memory (CM); and it can interrupt one central processor (CPU) program and initiate another. PPUs transmit absolute addresses to central memory, while the address transmitted by the CPU to central memory is biased by the contents of a Reference Address register and cannot exceed an upper bounds contained in a Field Length register. These registers are initialized whenever control of the CPU passes from one program to another. Transfers to and from ECS are controlled by the CPU: there is a Reference Address register and Field Length register for ECS as well as CM.

The peripheral processors contain system routines such as card reader, card punch, and printer programs, magnetic tape drivers, display programs, and a disk/drum executive. One of the peripheral processors contains a PPU Monitor, which controls the assignment of tasks to peripheral processors, allocates peripheral devices and data channels, and performs part of the job scheduling process. The PPU Monitor is also responsible for monitoring CPU jobs for time limit and time slice end.

There is also a CPU Monitor which is part of the central memory resident. The CPU Monitor performs such tasks as the allocation of ECS space, scheduling of central memory and the central processor, and transfers between CM and ECS. The CPU Monitor also processes requests from CPU programs for various system functions. The CPU Monitor is small in size and uses a series of overlays to perform its various functions: since a 500-word overlay can be loaded into CM from ECS in slightly less than 65 microseconds, the overhead cost is small. In addition to the CPU Monitor, the central memory resident contains pointers to the various tables contained in ECS, buffers for unit record devices such as printers and card readers, and an ECS-PPU communication area.

The 6000 series computers require rapid access to large volumes of data, and so most of these systems rely on some form of rotating mass storage device for storage of active files. A common characteristic of these devices is a high transfer rate coupled with a relatively slow access time. In view of this characteristic, it is generally desirable to transfer at each access the largest possible data block commensurate with the memory available for buffering and the anticipated

file and record sizes. In ECOS, almost all input/output is buffered through ECS, and so buffer sizes can be adjusted to balance the rate at which disk/drum requests are issued to the rate at which they can be serviced. One use of ECS, then, is to provide buffer space for sequential files: it is also possible that short files may exist entirely within ECS. As indicated earlier, ECS is also used to hold system tables and CPU Monitor overlays, and the system library is divided between ECS and rotating mass storage. A portion of ECS is allocated to the user programmer as auxiliary memory: the block allocated to a given program may be further divided in local and common areas for various sub-programs. FORTRAN contains statements by which the user can access this portion of ECS. Finally, ECS provides a staging area for job loading and roll-in/roll-out.

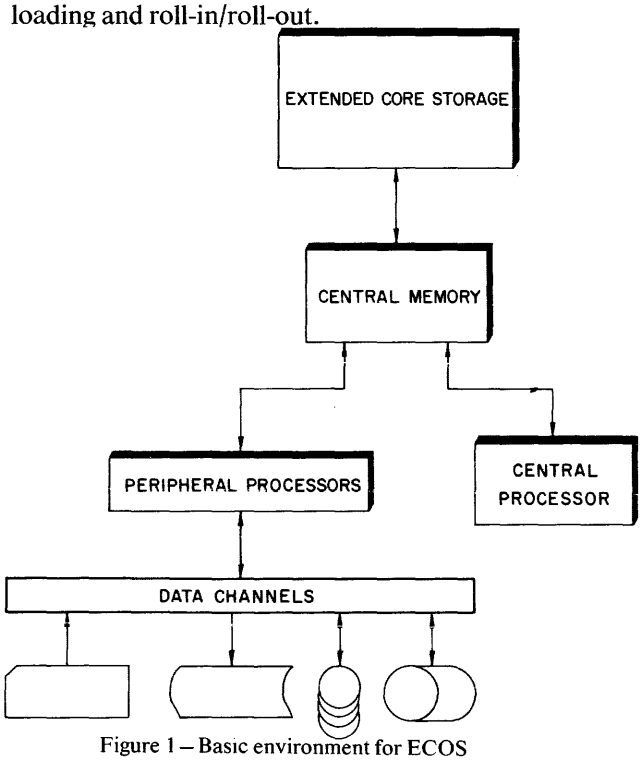


Figure 1 – Basic environment for ECOS

The foregoing description has necessarily been brief: it may become clearer if we follow a job through the system. Assume that the system has the hardware configuration of figure 1, and that our job enters the system via the card reader. The PPU Monitor periodically senses the status of the card reader and, if it finds the card reader in a ready state, it directs one of the peripheral processors to initiate a job loader program. The job loader inputs cards from the card reader and transmits them to a CM buffer. Control cards are examined during this operation and table entries made as required via requests to PPU Monitor. When the CM buffer is filled, or when the end of the job is encountered, PPU Monitor is notified. PPU Monitor, in turn, requests the CPU Monitor to move the contents of the CM buffer to an ECS buffer area. This requires interrupting the program currently using the CPU but, since this operation may require as little as 50 microseconds for a 256-word CM buffer, the effects of this interruption are negligible. When the ECS buffer is filled, the CPU Monitor constructs a disk/drum request and enters this request in a queue. This queue is scanned by a disk/drum executive program which resides in a peripheral processor. The disk/drum executive selects a request from this queue, requests CPU Monitor to move the desired data from ECS to the ECS-PPU communication area in central memory, and then transmits the data to the disk or drum.

The PPU Monitor is responsible for initial scheduling of the job. For example, if a job required magnetic tapes, the PPU Monitor would place a "hold" on this job until the specified tapes were available and ready. The ECS requirements for this job are then examined. All jobs are first loaded into ECS and from there loaded into central memory, so space in the ECS job staging area must be reserved as well as any programmer requirements for ECS. (Also, an ECS buffer is reserved for each file declared by the job.) When ECS space has been allocated to the job, the next control card is examined to determine what operation is to be performed next. Suppose our job requires a program which resides on the disk. The CPU Monitor inserts the disk request in the queue for the disk executive and, when the disk executive has transmitted a block to the PPU-ECS communication area in central memory, moves the program blocks to the ECS staging area for this job.

When the job is ready for execution, it is placed in a queue of jobs waiting for CM space. Once CM space has been allocated, the contents of the job's ECS staging area are rolled into central memory, and the central processor requested. When an I/O request is encountered during program execution, control is passed to the CPU Monitor. Suppose the operation is a read: the CPU Monitor checks the status of the buffer associated with the file being accessed and, if the buffer contains data, a block of data is transferred to the requester and control of the CPU returned. Should the buffer be empty, the CPU Monitor constructs a read request and places it in the request queue for the appropriate storage medium. If there is another job in central memory waiting for the central processor, it is given control of the CPU: if there is not, one or more of the jobs in central memory are rolled out into the ECS staging area, and a job from the queue of jobs waiting for central memory space is selected and rolled in. When a job completes, its ECS space is released, its buffers

are closed, and its print file(s) placed in a print request queue for subsequent processing by a peripheral processor program.

Simulator description: the job generator

If the workload of the system consisted of a single job or n identical jobs, prediction of system performance would be a comparatively simple task. In practice, jobs vary widely in their composition and characteristics, so it is necessary to deal with a *mix* of jobs, which is defined by a mean value and distribution. Studies have shown that certain job characteristics, such as compute time, may be approximated by an exponential distribution. Most often, the negative exponential distribution with density function ae^{-ax} has been used to fit job characteristic samples, although it appears that such samples might be better fitted by a hyper-exponential distribution because of the large variances encountered in practice. The ECOS simulator permits sampling either type of distribution.

The function of the Job Generator is to provide the simulated system with a mix of jobs. The mean values of various job characteristics are supplied to the simulator as parameters: the Job Generator uses these mean values to sample distributions and thus determines the characteristics of the individual jobs. The characteristics generated in this manner include the job's compute time, memory requirements, standard input and output file lengths, number of programmer files, programmer file length, and the number, if any, of programs to be loaded from the disk. The generated characteristics of the job are entered in a table. In executing simulator operations such as entering a job in a queue or manipulating a list of events, only the pointer to the table entry is manipulated: the table entries themselves are never moved.

Description of the simulator: system model and simulator structure

After generation, jobs in the system exist in one of several states until completed. These states, and the state-to-state linkage, are illustrated in figure 2 and described below. Each state in the operating system corresponds to a state processing routine in the simulator. Associated with each job's table entry is a state identifier and an event time. Each time a step in the processing of a job is performed, the time at which the next step is to be performed is computed and stored as the event time. The job is entered in a list according to this event time: when this point in simulated time is reached, the job is removed from the list and control transferred to the subroutine indicated by the state identifier.

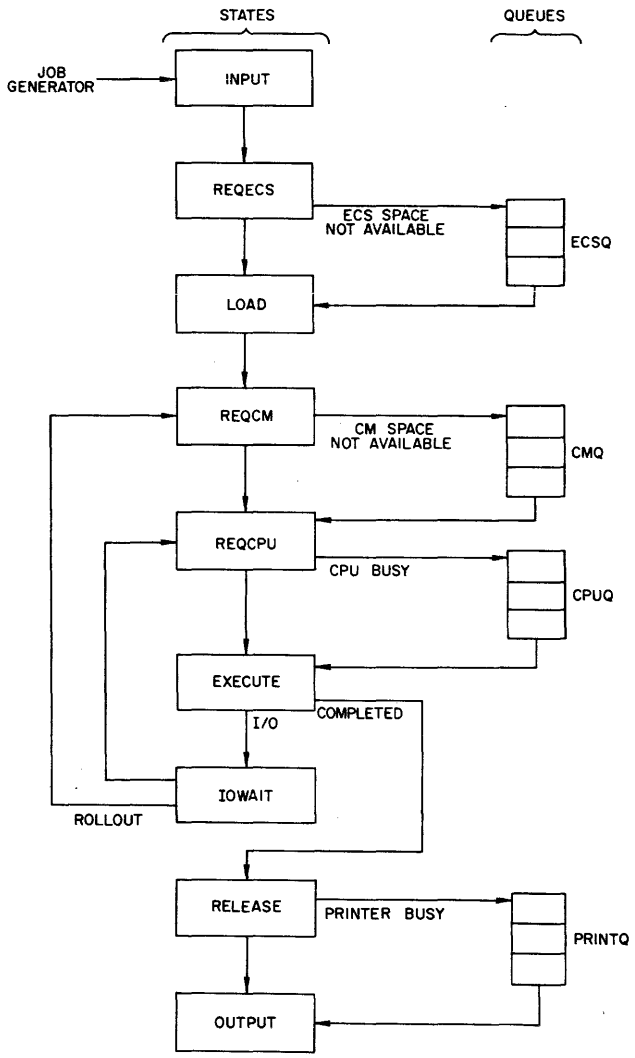


Figure 2 – ECOS model

When a state-to-state transition is scheduled but cannot be made (e.g., when a job requires the CPU but finds the CPU busy), the job is entered in a queue for subsequent rescheduling. Processing of queues and the list of event times is performed by a simulator routine called the Scanner, which will be discussed later.

State description

State INPUT — After a job is generated by the Job Generator, it is placed in the INPUT state, which represents the card-to-disk operation. The INPUT state routine examines a record count associated with the input file: if this count is non-zero, the routine decrements the count by an amount which depends on the size of the CM buffer and predicts, based on the card reader speed and buffer size, the time at which the buffer will be filled. (Note:

in the current version of the simulator, the card-to-disk operation is not buffered through ECS: instead, the contents of the CM buffers are transferred directly to the disk whenever the buffer is filled.) The event time for the job is set to this time, the state identifier is set to a negative value, and the job is placed in the event list. When the Scanner extracts from the event list a job whose state value is negative, it calls a disk simulator subroutine. This subroutine determines the time at which a disk request will be completed, sets the job's event time accordingly, sets the state identifier positive, and inserts the job in the event list. When the Scanner extracts a job from the event list which has a positive state identifier, it transfers control to the corresponding state routine.

A simulated job cycles in the INPUT state until the input record counter has been reduced to zero: it is then advanced to the REQECS state.

REQECS — If the available ECS space meets the job's requirements, then ECS job staging space and buffer space is assigned to the job, and the job is advanced to LOAD state. If the available ECS is not sufficient, the job is placed in a queue (ECSQ). Jobs in this queue are examined whenever ECS space is released: when space becomes available, the job is assigned space and advanced to the LOAD state.

LOAD — This state represents the loading of a program from the disk into the ECS staging area for the job. A counter is examined to determine the number of loads required: if nonzero, the state identifier is set nonzero to indicate a disk request, the load counter is decremented, and the job placed in the event list. A job cycles in LOAD state until the load counter is equal to zero, and is then advanced to the REQCM state.

This state represents the loading of programs from the library: a simulation parameter permits the disk requests to be all or partly replaced by ECS requests, thus simulating the effects of placing the system library in ECS.

REQCM — Simulation parameters specify the number of jobs which may simultaneously reside in CM as well as the size of CM. If sufficient space is available, and if the number of jobs currently in CM is less than the limit specified, CM space is assigned to the job and it is advanced to the REQCPU state. If either or both conditions are not met, the job is entered in a queue (CMQ). Jobs in this queue are examined whenever CM space is released and, if adequate space has become available, are assigned CM space and advanced to the REQCPU state.

REQCPU — If the CPU is not busy, it is assigned to this job and the job advanced to the EXECUTE state. If the CPU is busy, the job is entered in a queue (CPUQ). Whenever the CPU is released, a job is selected from this queue, the CPU is assigned to the job, and the job placed in EXECUTE state.

EXECUTE — Each job has a comparatively small buffer in CM for each file, together with a much larger buffer in ECS. (These buffer sizes are simulation parameters.) Based on the buffer sizes and on the file lengths generated for the job, the simulator determines the number of ECS-CM move operations and the number of disk operations for each job, together with the disk request issue rate. In the EXECUTE state, the CPU time for the job is examined: if it is zero, the CPU is released, the job's ECS and CM space is released, and the job is placed in the RELEASE state.

If the CPU time is non-zero, the interval of compute time before the next disk request occurs is predicted, and the CPU time decremented by this amount; also, the job's event time is advanced by this interval. The state identifier is set to a negative value, and the job is inserted in the event list. When the Scanner extracts a job from the event list which has a negative identifier, it calls a disk request to predict the I/O request completion time: also, if the request is from a job in EXECUTE state, the Scanner releases the CPU and places the job in IOWAIT state.

IOWAIT Jobs in IOWAIT state are suspended from the CPU and awaiting completion of a disk operation. On completion of this operation, the IOWAIT state routine places the job in the REQCPU state. However, if the Scanner finds the CPU idle and CPUQ empty, it may roll out a job from CM to the job's ECS staging area and change its state from IOWAIT to REQCM.

RELEASE The RELEASE state routing releases a job's ECS and CM space, and computes the time required to relocate CM and ECS. The job is then advanced to the OUTPUT state. If the printer is not busy, the job is entered in event list. If the printer is busy, the job is entered in a queue (PRINTQ) for subsequent printing of its output file.

OUTPUT This state represents the disk-to-printer operation. The Scanner periodically checks the status of the printer: if the printer is non-busy, the Scanner selects a job from PRINTQ and places it in OUTPUT state. The OUTPUT state routine examines a record count associated with the output file: if this count is non-zero, the routine decrements it by an amount which depends on the size of the CM buffer and predicts, based on the buffer size and printer speed, the time at which the buffer will become emptied. The event time for the job is set to this time, the state identifier is set negative, and the job placed in the event list. A job cycles in OUTPUT state until its output file record counter has been reduced to zero: it is then released from the system. (Note: in the current version of the simulator, the disk-to-printer operation is not buffered through ECS.)

Whenever an ECS operation (e.g., a data transfer, a job swap, ECS relocation, etc.) is simulated, the CPU time required for the operation is computed, accumulated, and added to the event time of the job currently in EXECUTE state.

Simulator description: the scanner

The various state routines are entered from, and exit to, a simulator routine called the Scanner. The Scanner is also responsible for the selection of jobs from queues for assignment to the system facilities and for selection of the next event from the event list. Execution of the roll-in/roll-out process is performed by the Scanner. The design of the Scanner routine permits changes in scheduling algorithms, queue disciplines, etc., to be readily made. The basic functions performed by the Scanner routine are illustrated in figure 3.

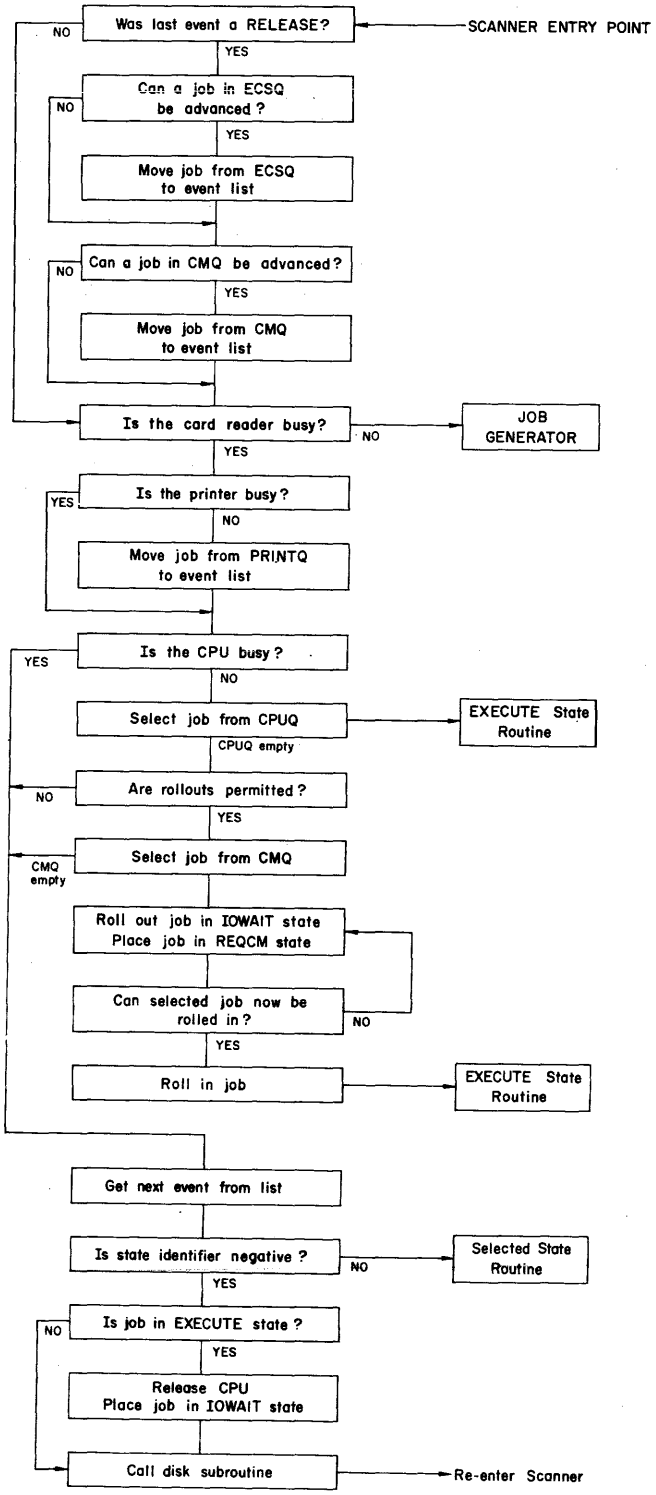


Figure 3 – Scanner routine for ECOS simulator

Simulator description: summary

In addition to the Job Generator, state routines, and Scanner, the simulator contains an input routine, a disk simulator routine, and a report generator. The input routine reads, lists, and interprets the parameter cards, while the report generator lists the results of the simulation. The disk simulator was designed as a separate package in order to permit experiments with different disks or with drums to be performed readily.

The organization of the main body of the simulator into state routines considerably eased the task of coding and debugging the simulator. This technique has several merits: it permits a modular design of the simulator with well defined communications between the various system routines, it simplifies the next event selection process, and allows the correspondence between system and model to be maintained at any desired level.

Some simulation results

In the study for which the simulation program was developed, it was desired to compare various ECS utilization strategies. In particular, it was desired to determine the incremental merits of:

a. using ECS to hold high-access library programs,
b. using ECS to provide large buffers for files (thus reducing the number of disk operations), and
c. using ECS to hold jobs during the loading phase and while waiting for the CPU or for an I/O operation completion.

A variety of configurations were simulated with job mixes varying from I/O-bound to compute-bound. One of the variables of interest was the CPU utilization (the percentage of total CPU time spent on users jobs). It was found that a configuration which yielded comparatively high CPU utilization even on jobs with a high I/O rate was a system which incorporated features (a), (b), and (c). In this system, up to six jobs were loaded into central memory (depending on the memory requirements on the jobs). When a job issued an I/O request which required a disk operation, control was transferred to another job, and so forth. If a point was reached at which all jobs were waiting for a disk operation, one or more jobs in central memory were rolled out into ECS, and a job waiting for CM was loaded.

Further experimentation with this system showed that reducing the number of jobs simultaneously residing in central memory had little effect on CPU utilization, since the added job swapping time was offset by the reduced CM relocation time. This led to the postulation of a system in which only one job resided in central memory at any one time. This job executed until it required a disk operation (or until the end of a time-slice was reached), at which time it was swapped with a job waiting in ECS. This concept was called *monoprogramming*.

Four system configurations were compared in detail; these were, in terms of the features described above:

1. BASIC—a multiprogramming system without ECS
2. BASIC + (a) and (b)
3. Monoprogramming (c)
4. Monoprogramming (c) + (a) and (b)

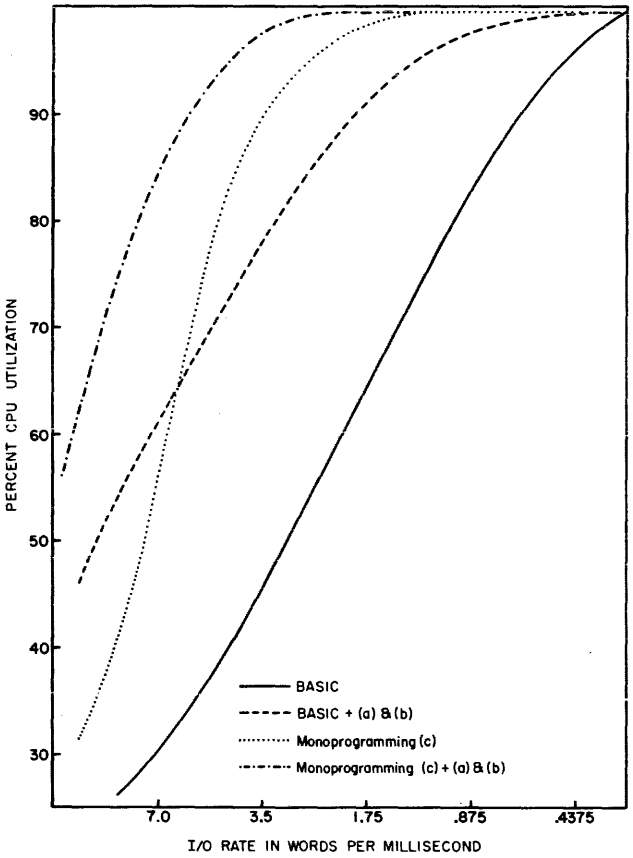


Figure 4—CPU utilization comparison

These systems were simulated for various hardware configurations: a typical set of results for a specific hardware configuration is shown in figure 4.

Future Plans

A 64/6600 system with ECS is a natural environment for an operating system designed to provide a large-capacity multi-access capability.

The multiprocessor design of the 6000 relieves the CPU of involvement in the monitoring of terminals, since this task can be capably performed by the peripheral processors. With ECS, the swapping time is sufficiently small (2.5 milliseconds to roll out one

10K job and roll in another) that overhead increases very slowly as the number of jobs being time-sliced increases. These factors minimize the direct effect of user activity on system overhead, reduce the complexity of the scheduling task, and lead to a multi-access system design which is general enough to be useful, efficient enough to be affordable, and realistic enough to be implemented.

In order to explore some of the facets of multi-access design, the ECOS simulator is being modified to provide for time-sliced scheduling of the CPU, and a remote terminal task generator is being incorporated.

ACKNOWLEDGMENT

The author is indebted to Pierre Chavy for guidance and assistance in the design and construction of the ECOS simulator.

REFERENCES

1 G J ALLEN
Extended Core Storage for the Control Data 64/6600 Systems this volume

2 J E THORNTON
Parallel Operation in the Control Data 6600
Proc FJCC 26 1964

A structural theory of machine diagnosis

by C. V. RAMAMOORTHY
Honeywell, Inc. - EDP Division
Waltham, Massachusetts

INTRODUCTION

The present trend in large scale integration of microelectronic technology has focussed a heavy emphasis on the maintenance and diagnostic aspects of large computers. Efficient techniques of diagnosis are important in multi-processors with reconfiguring capabilities to provide high availability. Also, a need exists for simple but effective means of understanding, visualizing and analyzing the probelms associated with diagnostics. This paper presents a unified approach based on graph theory, which seems to provide a new insight into the problem without regard to the level of detail under consideration.

The techniques developed here depend on the analysis and manipulation of system graphs represented by their connectivity matrices and hence implementable by computer programs.

The graph representation simplifies the understanding of the operation of a large system, and augments the ability of the maintenance engineer to cope with unforeseen and unexpected problems.

The theory proposed here does *not* pretend to solve all problems, but its value rests on the new insights it seems to provide to the machine designer and the diagnostic engineer. The task is incomplete and this paper is only a preliminary report on this subject.

Current efforts

We shall briefly summarize the current efforts in this area. The classical methods of diagnosis[1,2,3] classify the component elements into combinational and sequential entities. Various techniques are available for specifying a minimum set of tests to detect or locate physical malfunctions in non-redundant memoryless combinational circuits. In elements with memory property the theoretical determination of a minimal set of tests, either for fault detection or location, is complex. Where they exist, these procedures apply only to systems containing a small number of elements which often can be analyzed by exhaustive methods.

The black-box approach deals with input-output relationship of the system and develops efficient tests to verify a set of functional specifications. Since it ignores the machine structure, this method is not useful for fault location but could provide valuable acceptance tests. In the third approach,[4] the design is simulated under known component failure modes, the test results (called a test dictionary) are catalogued for further reference. Also, additional hardware may be added to facilitate the diagnostic function. This is a very popular method since the machine designer is not bothered with maintenance requirements initially and the diagnostic information is generated later as a part of machine simulation.

The fourth approach tailors the computer organization specifically for ease of maintenance and diagnostics. One such approach[6] partitions the machine into mutually exclusive subsystems, each having some capability of testing others, and in turn being tested by them. Such an approach may impose undue design constraints that could impede superior performance of the machine.

In the microelectronic technology the diagnosis must be tailored differently to the different phases in testing. For example, during on-line diagnosis, a fault must be pin-pointed at the level of the replaceable module (major board). The defective module then must be tested off-line and the malfunctioning element (flat pack) must be located for replacement.

Our basic goal in this paper will be to suggest techniques for the following:

(a) Structural representation of the system at an appropriate level.

(b) Partitioning or segmenting the system into a number of smaller subsystems purely from the structural description.

(c) Strategic location of test points for purposes of subsystem segmentation, isolation, injection and/or monitoring of data during diagnostic tests.
(d) Sequences in which tests must be performed for fault-detection and/or location.
(e) Determination of functional hard-core of the system.
(f) Discovering of conditions for subsystem self-diagnosability and explicit determination of those elements which are not self-diagnosable.

Representation

The choice of representation of a complex system depends primarily on the characteristics one wishes to study. Proper representation must mask out those details not pertinent to the problem at hand. It must be simple enough so that it can provide insight where needed. Functionally, the representation for the diagnostic process must be useful in areas of simulation, design automation and system fabrication.

Also, the method of representation must be uniform. For example, the same *type* of representation must be applicable to the sets (systems) as well as to their component elements (subsystems).

In this paper, we look at the machine from two distinct levels; the structural level and the behavioral level. A sequential machine can be specified purely from the behavioral aspects like input-output relationships. Once the machine is designed, its *structure* (interconnections between components and flow of information) comes into being. The composite machine can then be looked upon as the *superposition* of behavioral characteristics of the components on its structural form. This separation between behavior and form lends simplicity to the understanding but also sheds new light into the problems of diagnosis and maintenance.

We shall develop techniques that will analyze the given machine structurally, and develop information which can be used later with the behavioral criteria of the components to derive diagnostic procedures and also help in component assignment in modules and submodules (Figure 1).

Extension

Since graphs are used for structural representation, these techniques are applicable to computer systems, as well as to their programs.

Structural representation

The structure of the system is represented by the interconnection of its components. The components can be discrete logical elements like flip-flops and various

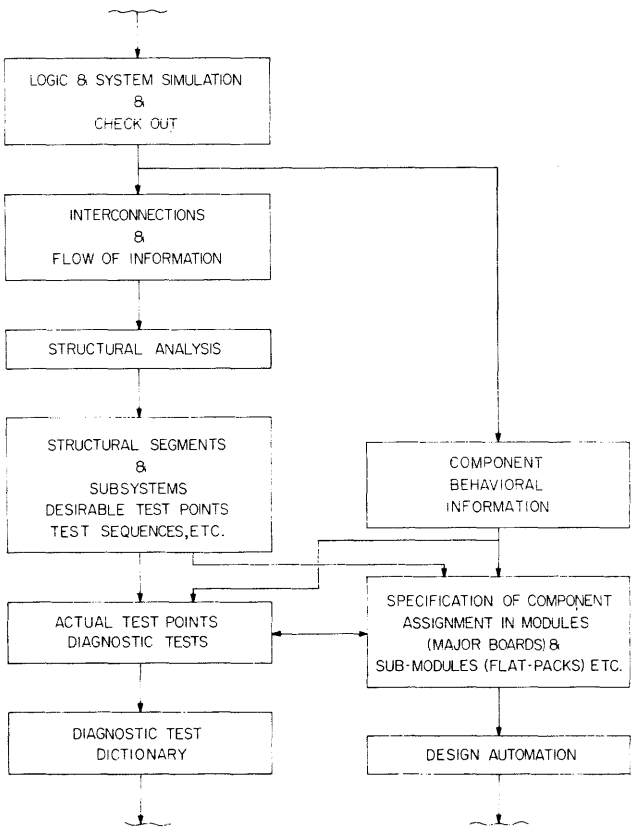


Figure 1—A proposed sequence of operations in design processing

types of gating, or functional units like counters and adders, etc. In the most practical case, the component may represent the *smallest replaceable functional module*, generally an integrated circuit major board or a flat-pack. In all subsequent discussions, we shall restrict ourselves to functional or logical elements which do not include those involved in power supplies, timing, etc.

Any discrete sequential system is isomorphic to a directed graph.[7,8,9] The nodes (vertices) represent functional elements (combinational and sequential) and the directed branches (arcs) represent lines of signal propagation. In particular, the *arc* from i to j describes the functional relation that the output of node i enters as an input to the node j. The computer system can be considered as a multi-level structure where each level can be analyzed in the same manner, since the graph representation is valid whether the node represents a logical building block or a complex functional module.

Let the system be represented by an n node graph G, with the node set {1,2,3 . . . n}. The *connectivity matrix* $C = \{c_{ij}\}$ of G is a $\times$ n matrix whose ij-th term $c_{ij} = 1$ if and only if there is a directed branch from node i to node j, otherwise $c_{ij} = 0$. The *reachability matrix* $R = \{r_{ij}\}$ of G is another n $\times$ n matrix

whose generic term $r_{ij} = 1$ if and only if there is at least one directed path (i.e., a concatenation of directed branches) leading from i to j. Basic construction techniques of deriving R from C are well-known.[7,8,9] Many properties of the graph and the discrete sequential system it represents can be studied generally by manipulations on the connectivity matrix of the graph.

A node p is *essential* in a graph with an initial node i and a terminal node j if and only if it is reachable from i, and j is reachable from it. The initial and terminal nodes are considered also essential. A graph consisting of a set of nodes and branches is said to be *strongly connected* if and only if any node in it is reachable from any other. *A subgraph of a given graph* is defined as consisting of a subset of nodes with all arcs (branches) between these nodes retained. A *maximal strongly connected (M.S.C.) subgraph* is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. All M.S.C. subgraphs are mutually disjoint (i.e., have no common nodes). Two subgraphs are said to be *unconnected or disjoint* if there is no arc from any node in one subgraph to another node in the other subgraph. A *link subgraph* is a subgraph that contains no strongly connected subgraphs or unconnected subgraphs in it.

The structure of the system and hence, the structure of its graph can be analyzed by certain operations on its connectivity matrix.[7,8,9]

Behavioral description

The behavioral description concerns itself with input-output characteristics of the elements of the system, and as such, is necessary for devising the diagnostic tests. Considering the system as a whole, this means carrying along a maze of detail. For this reason, we consider the structure of the interconnections and the flow of information first and derive valuable information before we use the behavioral description for devising the tests.

Testing sequences

If only the primary (externally controllable) inputs and observable (externally available) outputs are used for diagnosis, the total number of test sequences will be very large and the fault location will have low resolution. The size of the test dictionary will increase with the size of the structure and in particular, with the number of feedback paths and memory elements. Thus, segmenting or breaking up a large system into small subsystems is important to improve resolution, as well as to reduce the average time for test. Segmenting or partitioning is also important from another angle. The basic assumption made in deriving the conventional fault detecting and locating tests is that only a single fault is involved. This need not be true in all cases, since certain single faults are contagious in that they provoke malfunctions in adjacent regions. Segmenting, at least during the diagnostic phase, tends to break-up the system into mutually *non-interacting* subsystems and helps to diminish the effects of multiple faults.

Also, system segmenting can help in the component assignment problem in the design automation process (Figure 1).

Structural segmenting of a complex system

The first problem is to segment a large system into smaller subsystems. Any discrete sequential system as represented by its graph can be partitioned into its component maximal strongly connected (M.S.C.) subgraphs (subsystems) and link subsystems. Explicit algorithms are available to determine all the M.S.C. and link subsystems from the connectivity matrix of the whole system.[7,8] An example of the structural partitioning is illustrated in Figure 2a.

Basis graph

Each M.S.C. subsystem or link subsystem can be represented by a single node, possibly with multiple inputs or outputs. The system graph, after such a substitution, will contain less number of nodes than the original graph. We shall call this the *basis graph of the* system. The basis graph is a weakly connected graph, i.e., it has no strongly connected subgraphs. Figure 2b illustrates the basis graph of the system of Figure 2a.

Test points and test point pairs

Let t_i and t_j be two nodes in the system graph. The ordered pair (t_i, t_j) is defined as a compatible test point pair (or simply as a test point pair) if some input test sequence at node t_i can provoke a distinguishable output at another node t_j. The concept of the test pair can be extended to the case where each member of the ordered pair may be a subset of nodes, rather than a single node. In this case, the simultaneous application of test sequences into the primary input nodes provokes distinguishable outputs in the output nodes.

Without loss of generality, in the rest of the paper we shall consider only test point pairs with one input node and one output node for reasons of clarity and elucidation. The purposes of introducing test points are as follows:

(1) They can provide additional segmentation of a large subsystem for purposes of diagnosis, i.e., they can be used to isolate a selected subsystem from the rest of the system.

(2) They provide the means for injecting test inputs and/or monitoring the resulting outputs.

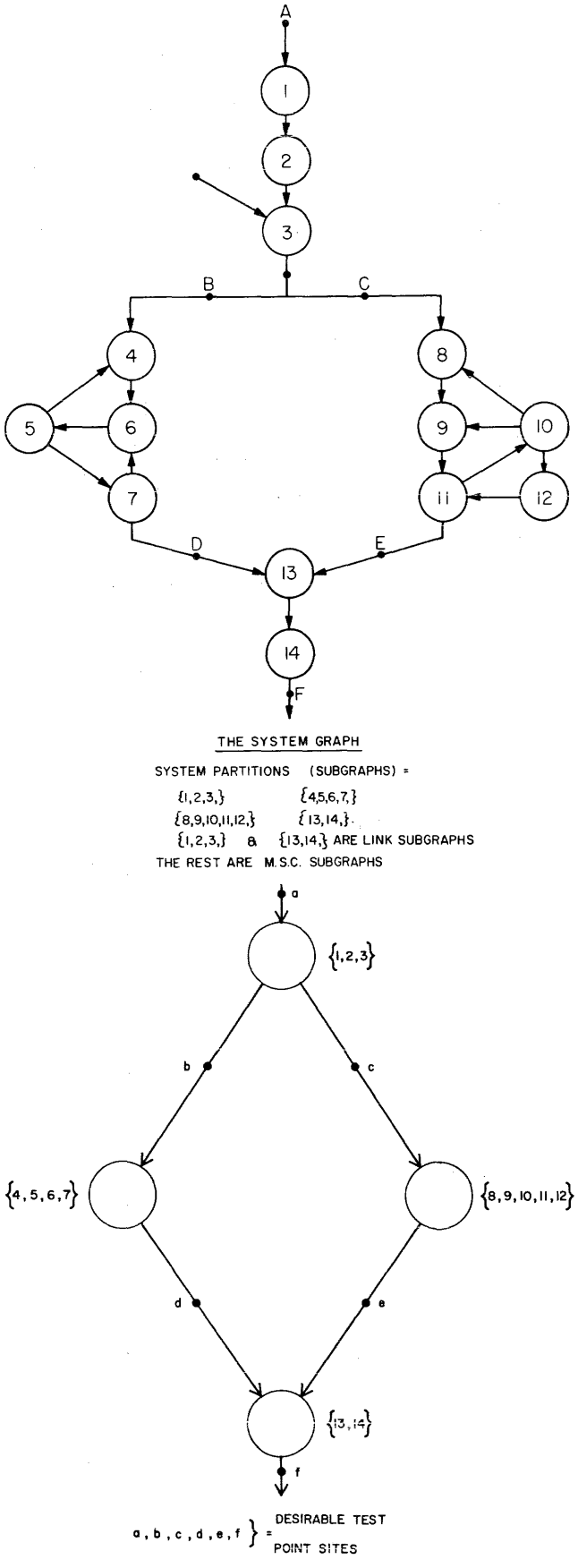


Figure 2(a,b)—A system graph

Determination of structural test points in a system

After the component subsystems are determined, it would be expedient to isolate them from the rest of the system during testing. This can be accomplished by inserting test points at strategic locations of the system. The test points should provide accessibility to other elements within the subsystem. A test point on a node can perform one or more of the following tasks:

(a) Disconnect the inputs of the node and provide entry for input patterns from an external source.

(b) Provide means for monitoring the outputs of the node.

When test point pairs are mentioned, it is assumed the input test point fulfills the first function and the output test point performs the monitoring function on the output.

The test points should be located at the entrances and exists of the subsystems which are derived from the partitioning of the total system. Thus, the test procedure will be to *isolate* the subsystem from its neighbors, inject the test sequences at the primary test points (entrances), and monitor the outputs at output test points (exits of the subsystem). Equivalently, the test points must be located at the entrances and exits of individual nodes in the basis graph of the system.

To isolate an M.S.C. subsystem or a link subsystem, it is only necessary that we determine its entrances and exits and insert "isolating" test points at these places. Since the exits of a subsystem become the entrances of the subsequent ones, the number of test points of this type will be at most Σ_i $(k_i + e_o)$ where k_i is the number of exits (entries) of the subsystem i, and e_o is the number of entry nodes into the system.

Given the system graph, the entrances or exits to its subsystems can be derived directly from its connectivity matrix.[7,8] The sites for desirable test points in the example are shown in Figure 2b.

Reducing the complexity of large subsystems

If some system segment (subsystem) is still too large, one needs other procedures for further segmentation. One possibility is selective "dissection" of the subsystem so that it "breaks-up" into still smaller subsets. This "dissection" is achieved by providing controlled "breaks" in the information (or signal) flow during system diagnostics.

Such "dissection" process also decreases the size of the test dictionary which can be defined as the fault detecting and locating procedures of the system. The number of test points may increase, however. The length of a test will, of course, vary with the subsystem behavioral complexity, particularly with the nature and extent of the memory and feedback elements.

The technique that we shall adopt for reducing the complexity of a large M.S.C. segment will be to reduce the number of feedback paths within it. In the graph-theoretic sense, the problem can be restated as follows: Given the graph of a discrete sequential system with all the primary input nodes (entrances into the system) and the exit (output) nodes specified, it is required to make the system an "open-loop" (loop-free) system with a *minimum* number of branch removals. (A minimum number is preferred purely from cost considerations.) The most important *constraint* is that the branch removals must be such that all the nodes in the subsystem be *reachable* from the primary input nodes or points. We shall present an algorithm for this purpose.

STEP 1: Segment the system into M.S.C. and link subsystems by procedures outlined previously.[7,8]

STEP 2: For each segment in Step 1, determine the entry nodes. In the case of the M.S.C. subgraphs, the procedure to determine the entry nodes are as given in Refs. 7 & 8: For link subgraphs, these procedures are not necessary, for obvious reasons.

STEP 3: Take the entry node of an M.S.C. subsystem. Delete or disconnect all the directed branches *entering* it. Record the branches thus disconnected.

STEP 4: Partition the altered subgraph into any M.S.C. and link subgraphs.

STEP 5: If there are any M.S.C. subgraphs present, iterate Steps 2 through 4. If none, the system is loop-free and note the number of branches removed. They are the minimum number of branches that should be removed to make the system open-looped with all nodes reachable from the primary or entry node.

The above procedure assumes that there is only one entry node in the M.S.C. subgraph and subsequent modifications successively create single entry nodes. Since, in general, the number of entry nodes may be more than one, we modify the procedure slightly. In this instance, we wish to select that entry node into the subsystem which results in an open-looped system requiring a minimum number of branch removals. For this, we select each primary input node in succession and apply Steps 2 through 5 and determine the number of branches removed for the choice. We select that open-looped system which results in least number of branch removals. We note the identification of the branches removed during the process. (It is easy to implement controlled disconnection of branches by logical means.)

Comment: Even though recursiveness of the above procedure promises programming simplicity, the number of computations can be significantly reduced by the following modification of the procedure: When there are multiple primary entry nodes, we select the node with the *largest* branching ratio which is the quotient obtained by dividing the number of emanating branches by the number of incident branches into that node. There is a heuristic justification to this procedure and it is good for large systems where a quasi-optimal solution is adequate.

In Figure 3a we have selected a simple M.S.C. subgraph for open loop reduction. Since node 1 is the primary input node, we break the feedback path (3,1) first. This makes nodes 2, 3, and 4 strongly connected with the primary input coming into node 2. We break feedback path on node 2, viz., (3,2). The graph is open looped with the primary input at note 1 reaching all nodes.

Auxiliary test points

The process of breaking up the loops in a closed loop system may sometimes introduce nodes whose outputs must be monitored by auxiliary test points. The locations of auxiliary test points can be found explicitly from the connectivity matrix of the system as follows:

Let C' be the connectivity matrix of the open-looped sequential system. Let {e} be a subset of its nodes which are exits from the system. The auxiliary test points are the outputs of those nodes with all-zero column vectors and which are not the exit nodes of the system. In the case of Figure 3b, the site of the auxiliary test point is the node 3.

Properties of test point pairs

The necessary condition that (t_i, t_j) be a test point pair is that t_j is *reachable* from t_i. This implies that there exists at least one path from node i to node j. The *range* of a test point pair (t_i, t_j) is defined as the subset of nodes which can react to the test input at the input test node t_i, and provoke an output possibly distinguishable and monitorable at the output of node t_j. Given the connectivity matrix C (dimension $n \times n$) of the system and (t_i, t_j) be a test point pair in the system, then the range $\gamma(t_i, t_j)$ of the test point pair is explicitly determined by the non-zero elements of the vector:

$$\gamma\ (t_i, t_j) = [R_{ti} \cap R^T_{tj}] \cup [e^{ti} \cup e^{tj}]$$

where

$R_{ti} = t_i$-th row of the reachability matrix of C.

$R^T_{tj} = t_j$-th column of the reachability matrix of C and e^k is a binary row vector of dimension n such that only the k-th element is a "one".

In other words, if p-th column of $\gamma(t_i, t_j)$ is a one

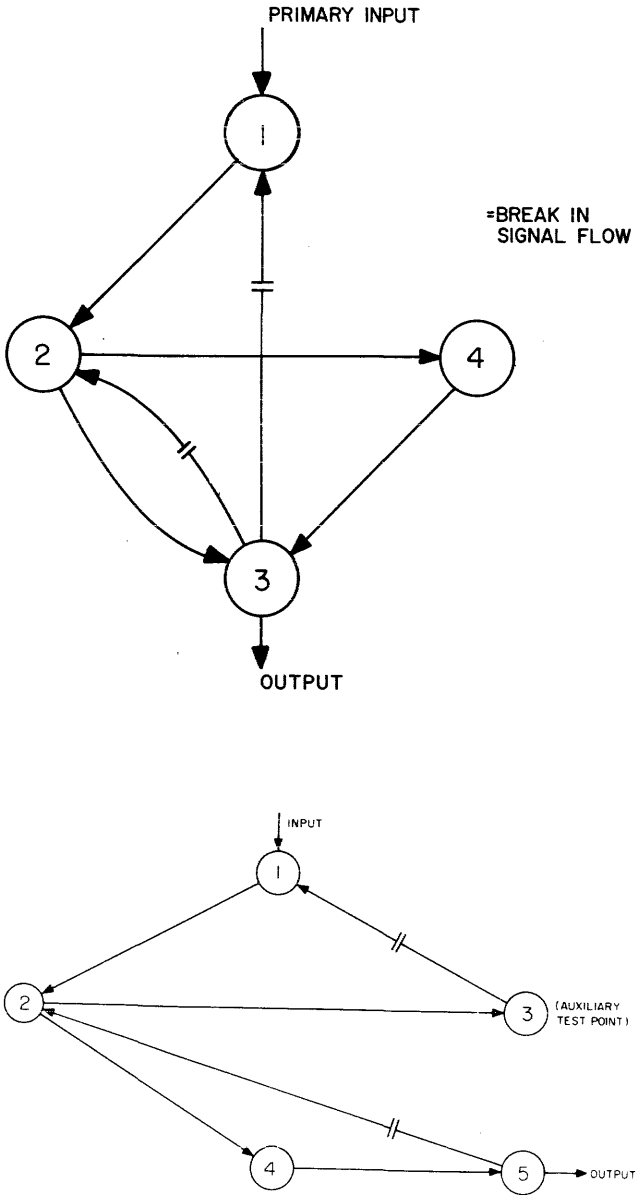


Figure 3(a,b)—Breaking up feedback paths

implies that node p is in the range of test point pair (t_i, t_j). (See Figure 4 for example.)

The proof of this can be sketched as follows: Since (t_i, t_j) is a compatible test point pair, there is a path from t_i to t_j. The non-zero elements of R_{t_i} correspond to all those nodes that *can be reached* from test input node t_i. $R^T_{t_j}$ is a vector whose non-zero elements that correspond to nodes that *can reach* the test output node. $(R_{t_i} \cap R^T_{t_j})$ are those nodes which are influenced by the test inputs at t_i and whose outputs in turn influence the output of t_j. e^{t_i} u e^{t_j} are the node vectors corresponding to the test input and output nodes.

We can readily determine the range vector for the generalized case when the input and the output members of a test point pair contain more than one node. Thus, $\gamma(\{t_i\}, \{t_j\}) = \gamma(\{t_{i_1}, t_{i_2} \ldots t_{i_{n_i}}\}, \{t_{0_1} t_{0_2} \ldots t_{0n_0}\})$

$$= \bigcup_{\substack{j=1 \\ k=1}}^{\substack{k=n_0 \\ j=n_i}} \gamma(t_{i_j}, t_{0_k})$$

where $\bigcup$ is an extended union operator.

We now state the following theorem without proof: A set of test point pairs can completely test the system if and only if every system element (node) is in the *range* of some test point pair $(t_i, t_j) \in T \times T$ where T is a set of all test points.

This theorem helps to select a minimum number of test points to check the system. It also states a less obvious but important fact. Even though the test points are selected based on their range over the nodes of the system, they also test all the interconnections (branches) between the elements.

Selection of optimum number of test point pairs for fault-detection and the analogy to prime implicant tables

Since the test points determined above may not all be necessary for fault detection, a minimum set of test point pairs can be determined by an algorithm which is analogous to the one used in the prime implicant selection in the simplification of Boolean functions.[10] We shall next state this algorithm.

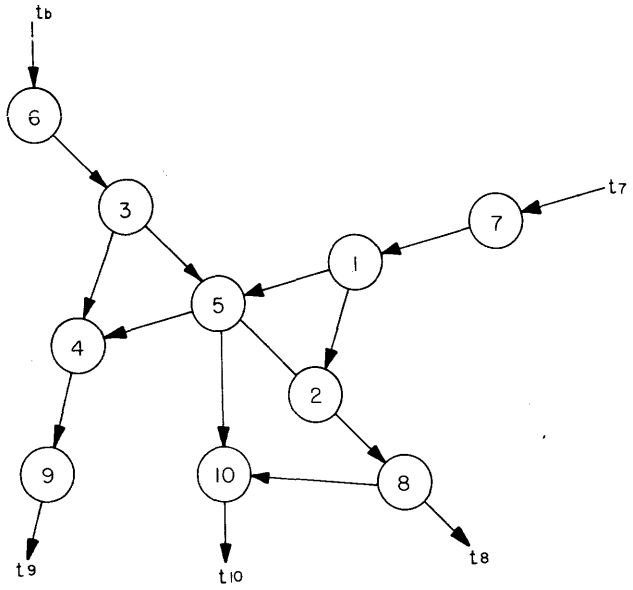
Given a list of desirable test point pairs derived from structural decomposition and the behavioral considerations of the component elements, the ranges of individual test point pairs can be derived explicitly by the methods suggested earlier. Their total range (the union of all individual ranges) should encompass all the elements in the system, since otherwise the testing would be inadequate.

Let the system contain n elements (nodes). Let there be p test point pairs in the system. The range of each test point pair can be indicated by a row vector of dimension n.

A Boolean matrix of dimension p x n can be developed such that its i-th row represents the range of i-th test point pair. For *fault-detection,* we wish to find the smallest set of rows such that there is at least a "one" in every column amongst the selected rows. This problem is identical to the one of selecting prime implicants in the simplification of Boolean functions. Since the number of test points in a system are generally much smaller than the number of elements in the system, McCluskey's algorithm[10] can be effectively used.

However, for systems with large numbers of test point sites, a near optimal selection with considerably less computational complexity is possible by the heuristic algorithm given in Appendix I.

An example of the test-point site selection for fault-detection is given:



$Y(t_6,t_8)=(\underline{R}_6 \cap \underline{R}_8^T) \cup (\underline{e}^6 \cup \underline{e}^8)$

$\underline{R}_6$ = REACHABILITY VECTOR OF NODE 6
= (0111100111)

$\underline{R}_8^T$ = REACHING VECTOR OF NODE 8
= (1110111000)

$\underline{e}^6$ = (0000010000)

$\underline{e}^8$ = (0000000100)

$Y(t_6, t_8)$ = (0110110100)

Figure 4—A subsystem graph

The test point pairs in the system of Figure 4 are (t_6, t_8), (t_6, t_9), (t_6, t_{10}), (t_7, t_8), (t_7, t_9) and (t_7, t_{10}). In Figure 5, the "prime implicant"matrixisgiven in which the rows represent the range vectors of test point pairs and the columns the nodes of the system. Applying the well known methods of solving prime implicant tables, the complete detection scheme will only require the test point pairs (t_6, t_9) and (t_7, t_{10}).

TEST POINT PAIRS	RANGE VECTORS SYSTEM ELEMENTS (SEE FIG. 4)									
	1	2	3	4	5	6	7	8	9	10
$A = t_6 - t_8$		1	1		1	1		1		
$B = t_6 - t_9$			1	1	1	1			1	
$C = t_6 - t_{10}$		1	1		1	1		1		1
$D = t_7 - t_8$	1	1			1		1	1		
$E = t_7 - t_9$	1			1	1		1		1	
$F = t_7 - t_{10}$	1	1			1		1	1		1

Figure 5—Range matrix

Fault locating by using multiple test point pairs and the determination of test point pairs diagnosing a system element

Given a system with a number of test point pairs, it may be of interest to determine those test point pairs that influence a particular system element. The algorithm that does this is given as follows:

Let C and R be the connectivity and reachability matrices of a system with n elements. Let the elements be numbered 1 through n. Let the nodes which are also test points be given by another n element vector t.

Let the element to be *diagnosed* be given by "a": $a \in \{1, 2, 3, \ldots n\}$.

1. The test points whose inputs can influence the output of the system element are the non-zero components of the row vector $t \cap (R^T)_a$.
2. The test points which are influenced by the output of the system element are the non-zero columns of the row vector $t \cap (R)_a$.
3. The test point pairs that can diagnose the system element "a' are those ordered pairs (t_i, t_j) where t_i and t_j are non-zero columns of $t \cap (R^T)_a$ and $t \cap (R)_a$ respectively, and such that t_j is reachable from t_i or equivalently, $R_{t_i\ t_j} = 1$.

A tabulation of test point pairs that can diagnose specific system elements is useful in compiling the test dictionary, as well as in locating faults.

To illustrate with an example, the system element 3 in Figure 4 is diagnosed by test point pairs (t_6, t_9), (t_6, t_{10}) and $t_6, t_8)$, whereas the element 5 by (t_6, t_9), (t_6, t_8), (t_6, t_{10}), (t_7, t_{10}) and (t_7, t_8).

Structural resolution and indistinguishability at a system element

We shall define the structural resolution at a given element as the total number elements indistinguishable from it from a structural testing viewpoint. We shall call node a and node b indistinguishable if and only if both "a" and "b" lie on identical test ranges and none other. To discover the indistinguishable elements at any specific element "i" we proceed as follows:

(1) Let (t_{11}, t_{12}), (t_{21}, t_{22}), . . . (t_{p1}, t_{p2}) be test point pairs and let $\gamma_1, \gamma_2, \gamma_3 \ldots \gamma_p$ be their range vectors respectively.

(2) Let the given element "i" correspond to the i-th column in the range vector. Develop the transformed range vector γ_k^t for all k such that $\gamma_k^t = \gamma_k$ if the i-th column of γ_k is a "1", and $\gamma_k^t = \gamma_k$ (complement vector of γ_k) if otherwise.

(3) Perform a logical multiplication of all the transformed range vectors. The non-zero components of the logical product vector correspond to those elements structurally indistinguishable with the given element "i".

As an illustration, in Figure 4 the indistinguishable element corresponding to element 8 is element 2, since the logical product of the transformed range vectors is (0100000100). The concepts of structural indistinguishability and resolution are important in the location of test points.

A combined fault detecting and locating procedure

We shall next present the sequence in which the system elements must be tested for fault detection and location.

Three procedures are developed. The first is designed for debugging newly built equipment in which each step depends on the correct operation of elements tested previously. The second procedure is useful for maintenance of newly developed equipment, when no reliable failure statistics are available. The third is used when component failure statistics are available and the sequence of tests are optimized to minimize the average number of tests or their cost.

Fault detection and location at the subsystem level during debugging

Let the system be represented by its basis graph. Each node in it will be called a subsystem. Let n be the number of nodes in the basic graph. Let us assume that there are test points at the entrances and exits of these subsystems. (This assumption is not necessary except for the sake of exposition.)

Let C be its connectivity matrix. We determine the *columns* in C matrix with all zeros. Let S_1 represent the set of these nodes. We *delete* the *rows* and *columns* of C matrix corresponding to the nodes in S_1. We determine the columns in the remaining matrix whose components are all zeros. Let S_2 be the set of these nodes. We then delete the rows and columns corresponding to these nodes. We repeat the procedure, modifying the C matrix iteratively, until there are no more rows or columns left.

Let S_i be the set of nodes corresponding to all-zero columns of the C matrix during the i-th step. Let the set of nodes in the basic graph be $S = \{1, 2, 3, \ldots n\}$. Then the subsets $S_1, S_2, \ldots S_i, \ldots S_p$, obtained in the above procedure are precedence partitions on the set S.

The sequence of the testing of the nodes must be in the order of $(S_1, S_2, S_3 \ldots S_p)$. This means we test the nodes corresponding to set S_1 first, then if they are faultfree, we test the set S_2 (which may depend on the outputs of S_1) . . . and so on, through S_p.

If S_{i-1} is fault-free and S_i is faulty, it implies that S_i is faulty and all the subsystems S_1 through S_{i-1} are fault-free. Note that this procedure is applicable to any number of faults within the system, except of course, the hard core.

Since the basis graph is not a strongly connected graph, this procedure will always work. We shall illustrate this by an example (Figure 6).

EXAMPLE:

C =

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	1	0	1	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	1	0	1	0
8	0	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0

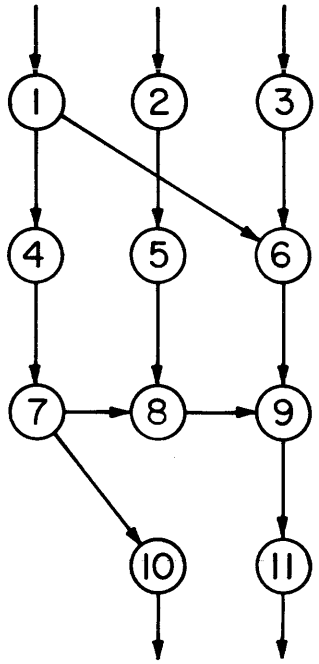


Figure 6—A system graph

Precedence Partitions or
= ({1,2,3}, {4,5,6}, {7}, {8,10}, {9}, {11}).
Sequence of all zero columns removed
Sequence in which tests must be conducted are: (1,2,3), (4,5,6), (7), (8,10), (9), (11).
Note that nodes 1, 2 and 3; 4, 5 and 6; 8 and 10; can be tested in parallel

Fault detection and location in newly developed equipment

An elegant fault detection and fault location sequence can be derived by reversing the sequence of tests in the previous procedure. We assume here that the probability of the system being faulty is very low, and it is expedient to test the whole system first for fault detection and then proceed on to fault location tests only if the former fails. Also, it is assumed that no failure statistics are available.

On detection of failure, we keep reducing the area under test successively after each test until a fault-free condition is detected. This transition indicates the location of the failure.

The example in Figure 6 is treated using the new procedure below:

Example: (Assume a fault in node 4)

Test all nodes using the primary inputs at nodes 1, 2 and 3 and the outputs at nodes 10 and 11. Since a fault is detected, we monitor the output of node 11; then 9; then 8 and 10; etc., using the primary inputs at nodes 1, 2 and 3. The output of node 4 indicates a fault, but no fault is found on nodes 1, 2 and 3. We thus ascertain that the fault is at node 4. However, this procedure is inferior to the next procedure to be discussed with the assumption that the failure probabilities of subsystem partitions are equally likely.

Fault detection and location tests where element failure probabilities are known

Let the system contain n subsystems, i.e., the nodes of the basis graph. We next perform the precedence partitioning of the system as indicated in a previous section. Let the sequence of partitions be S_1, S_2 . . . S_p. Corresponding to each partition we compute its a priori probability of failure, from the failure probabilities of its component elements. Thus, we shall assume that we know P_i, the a priori probability of failure of the subsystem partition S_i. P_i's are also as-

sumed to be statistically independent, discrete constants and $\Sigma P_i = 1$

In Figure 7, the subsystem partitions S_i's and their failure probabilities P_i's are shown in sequence.

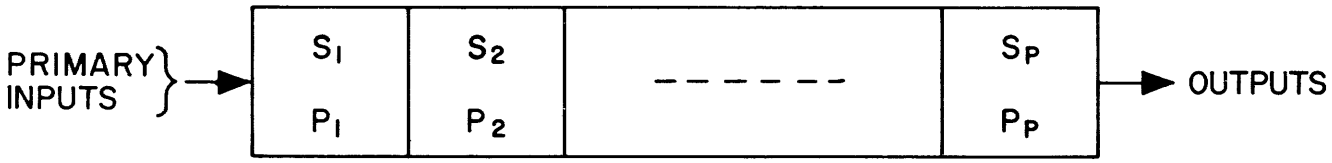


Figure 7—Precedence partitioning

We first perform the fault detection test by using the primary inputs and outputs I and if fault is found, we go to the fault locating test as given below.

Given S_i and P_i, it is required to develop a sequential "search" procedure that would locate the sybsystem partition containing the bad element, such that the average number of tests is a minimum. The procedure is as follows:

STEP 1: Find the subsystem partition S_k in Figure 7, such that

$(P_1 + P_2 + \ldots P_k) \geqq (P_{k+1} + P_{k+2} \ldots\ldots P_p)$ and

$(P_1 + P_2 + \ldots P_{k-1}) < (P_k + P_{k+1} \ldots\ldots P_p)$

We test the output of the subsystem partition S_k using the primary inputs and if a fault is detected then it must be in a partition earlier in precedence to S_{k+1}, i.e., it must be within $S_1, S_2 \ldots S_k$. We take step 2a. If no fault is detected at the output of S_k, then the fault must be in the partitions $S_{k+1}, S_{k+2} \ldots S_p$. We take step 2b.

STEP 2a: Find the subsystem partition S_{k_1} such that

$(P_1 + P_2 + \ldots\ldots P_{k_1}) \geqq (P_{k_1+1} + P_{k_1+2} \ldots\ldots P_k)$

$(P_1 + P_2 + \ldots\ldots P_{k_1-1}) < (P_{k_1} + P_{k_1+1} \ldots\ldots P_k)$

and

Use the output of the subsystem partition S_{k1} with primary inputs at S_1 to test the system. If a fault is detected, perform an iterative procedure similar to Step 2a on the faulty partitions. Otherwise, take the iterative procedure similar to Step 2b.

STEP 2b: Find the subsystem partition S_{k_2} such that

$(P_{k+1} + P_{k+2} \ldots\ldots P_{k_2}) \geqq (P_{k_2+1} + P_{k_2+2} \ldots\ldots P_p)$

and

$(P_{k+1} + P_{k+2} + \ldots P_{k_2-1}) < (P_{k_2} + P_{k_2+1} \ldots\ldots P_p)$

Test the system using primary inputs at S_1 and the outputs at S_{k_2} If fault is detected, then it is amongst $S_{k_2+1}, S_{k_2+2} \ldots S_{k_2}$ so that we take an iterative step similar to 2a. Otherwise, the fault is amongst S_{k_2+1}, $S_{k_2+2} \ldots S_p$ in which case we use a step similar to 2b.

This iterative procedure will stop when the subsystem partition containing the bad element is located.

Reference 11 has shown in a different context that the above procedure is almost optimal when the cost of performing the test for each precedence partition is the same. Modifications to the above scheme may be needed when test cost varies with different partitions.

Example:

In the example of Figure 6, let the a priori failure probabilities be given by the following table

Partition	S_1	S_2	S_3	S_4	S_5	S_6
Subsystems in the Partition	1,2,3	4,5,6	7	8,10	9	11
a priori probability of failure	.3	.2	.3	.1	.05	.05

Applying the algorithm in the previous section, if a failure is detected on the total system, we select the output of S_2 for the next test; if the test is bad, we test the output of S_1; if it is bad, then S_1 is faulty; otherwise, the failure is located in S_2 which contains subsystems 4,5 and 6. Since they are tested in parallel, the individual faulty subsystem can be distinguished. We now give the sequence diagram for testing for the above example (Figure 8).

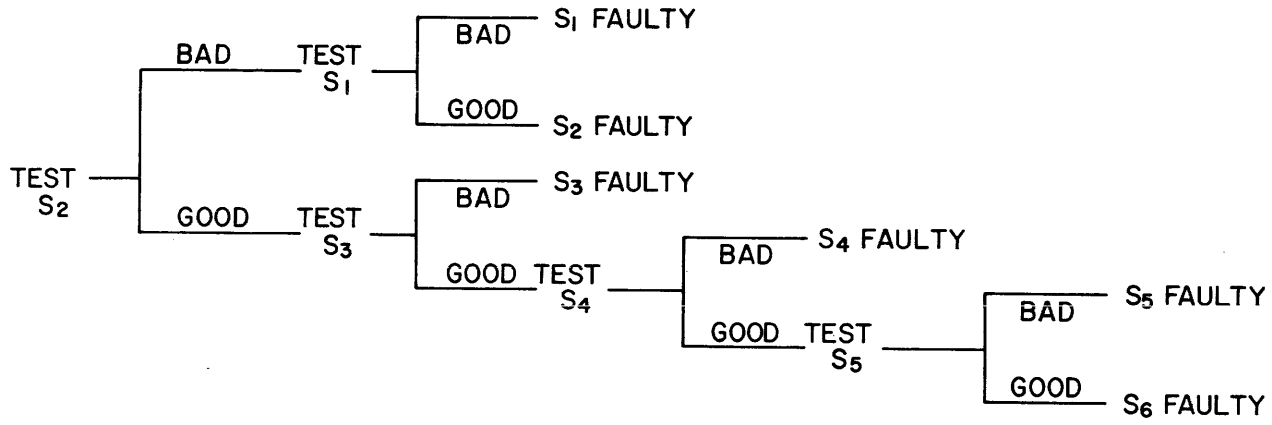


Figure 8—Test sequence for system in Figure 6

The average number of fault location tests per subsystem is 2.3. If the a priori probabilities subsystem partions are all the same, then the test sequence follows a simple binary search pattern with an average number of tests per subsystem = 2.67. If the test procedure suggested for fault location when no fault statistics are available is used, the average number of tests would be 4.45, which indicates that the procedure, based on the third technique with equal a priori probability assumption, is superior.

Fault location within a subsystem

After the fault is isolated at the subsystem level, it may be necessary to locate it at the next lower level (replaceable unit level). This is achieved in two ways:

(a) By performing an open-loop transformation of the system and using procedures discussed before, and/or

(b) By using the range intersection technique described below:

Range intersection technique

Let there be p test point pairs (t_{11}, t_{12}), $t_{21}, t_{22})$ . . . $t_{p1}, t_{p2})$ where t_{ij}'s are not all distinct. Let their ranges be given by $\gamma_1, \gamma_2, \ldots \gamma_p$, respectively, where γ_i is range vector of the test point pair i. The intersection (logical product) of vectors γ_i and γ_j is another vector whose non-zero columns correspond to the nodes common to both the ranges. Thus, if a test on γ_i fails and γ_j succeeds, the elements in trouble are those that correspond to node vector $(\gamma_i \cap \gamma_j)$ which is a subset of set γ_i. Knowing the membership of the vector $\gamma_i \cap \gamma_j$, a finer resolution of the fault can be obtained by selecting another range γ_k which has some nodes in common with node vector $(\gamma_i \cap \gamma_j)$. Thus, the test dictionary that is compiled on the range information can be used here. An example follows:

Suppose the element 4 in Figure 4 is faulty. Then the tests due to test point pairs (t_6, t^8) and (t_6, t_{10}) will be successful, but the tests due to other test point pairs will be unsuccessful. Thus, $\gamma(t_6, t_9) \cap \gamma (\cap \gamma(t_7, t_9) \cap \bar{\gamma} (t_6, t^8) \cap \bar{\gamma}(t_6, t_{10}) \cap \gamma(t_7, t^8) \quad \gamma (t_7, t_{10})) = (0001000010)$, which indicates that the elements 4 and 9 can be faulty. It is not possible to increase the resolution structurally beyond this.

Hard core and self-diagnosability

The hard core is defined as that part of the system which must be fault-free to allow an automatic test procedure to run and interpret the first experiments upon the system. The diagnosis of the hard core must be performed manually and hence, the system design must try to minimize the extent of the hard core or incorporate protective redundancy in it.

Even though power supplies, timing, etc., are part of the hard core, from a functional point of view the hard core must consist of some memory and arithmetic capabilities.

Considering the system in Figure 6, the first set of nodes corresponding to precedence partition S_1 (nodes 1, 2 and 3) must be operational before nodes of the next set S_2 (nodes 4, 5 and 6) can be tested. Thus,

the equipment that is required to test S_i, viz., the test input generators for node set S_i, the output test points which monitor the outputs of these, and test-administrating and evaluating, functions and equipment are the functional hard core of the system.

When the hard core of the system has been determined to be fault-free, the known operable condition of a machine can be expanded outward from the hard core by a proper sequence of tests. The condition for self-diagnosability is that each subsystem (excluding the hard core) must be diagnosable by other subsystems. The rationale behind this is as follows: A completely self-diagnosable system is preferred since this may reduce the need for many test points and their external control. However, this may imply excessive hardware since each diagnosing subsystem must have some arithmetic (logical) and storage capabilities. Two problems arise in the self-diagnosable systems:

(a) Determination of diagnosable subsystems, and
(b) The sequence of tests on the total system.

The determination of diagnostic subsystems will mean the fulfillment of the following conditions:

(a) The existence of a hard core and its ability to to diagnose a specific diagnostic subsystem.
(b) Each diagnostic subsystem can be diagnosed by one or more diagnostic subsystems.

It is obvious that the diagnosing and object subsystems must have no system element in common. Once the determination of the diagnostic subsystems are made, a system graph, based on the diagnosability relation, can be derived.

Let us now consider a self-diagnosable system and its component elements[6]. The branch (i,j) implies that the diagnostic subsystem j is diagnosed by another diagnostic subsystem i. The system graph is given in Figure 9.

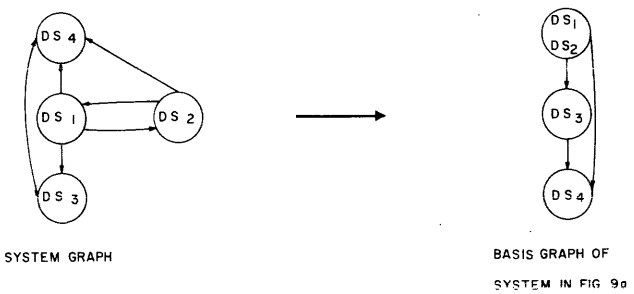


Figure 9(a,b)—System graph

Using the first test proposed earlier in the section on combined fault detection and location, the test sequence will diagnose the diagnostic subsystems DS_1 and DS_2 first, then will test DS_3 and finally DS_4.

We also note that due to strong connectivity of DS_1 and DS_2, they cannot be tested individually. Here the "open-looping" techniques discussed earlier may be useful.

Determination of diagnostic subsystems and the requirements for self-diagnosis

The necessary condition that a subsystem S_j diagnoses another subsystem S_i is that all the inputs coming into S_i must be controllable and all the outputs going out of S_i must be monitorable by S_j. This can be achieved by means of test points as discussed before, or by requiring that the diagnostic subsystem S_j have the capability of linking with S_i and initiating and evaluating tests, directly by itself, or through indirect access.[6] One can use these conditions to partition a given system into diagnostic subsystems. From the connectivity viewpoint, S_j diagnosing S_i implies that S_j gets itself strongly connected with S_i. We can say that a diagnostic subsystem and its object subsystems are members of the same maximal strongly connected subgraph.

To determine the diagnostic subsystems within a given system, we first partition the system into its M.S.C. subgraphs and the weakly connected elements. Next, we examine each M.S.C. subgraph to see if some of its elements can perform diagnostic functions on the other elements. If so, the M.S.C. subgraph is labeled as a diagnostic subsystem. A diagnostic subsystem used in this context represents both the diagnosing and the target subsystems. There is also a node (a M.S.C. subsystem) which is the hard core. The condition for self-diagnosability is that there should exist a sequence of nodes stretching onwards from the hard core so that each node (whether a diagnostic subsystem or not) can be tested. In other words, the whole system must be strongly connected as a chain. If this is not the case, additional data paths (branches) and their control would be needed to provide this condition. There may exist many alternative data paths that can also give the self-diagnosability. Behavioral characterization may be important in selecting appropriate data paths. We shall illustrate by an example (Figure 10).

The elements 1 & 2 are diagnostic subsystems. Data paths from node 2 to node 1, node 4 to node 2, and node 8 to node 1 are sufficient to make the system self-diagnostic.

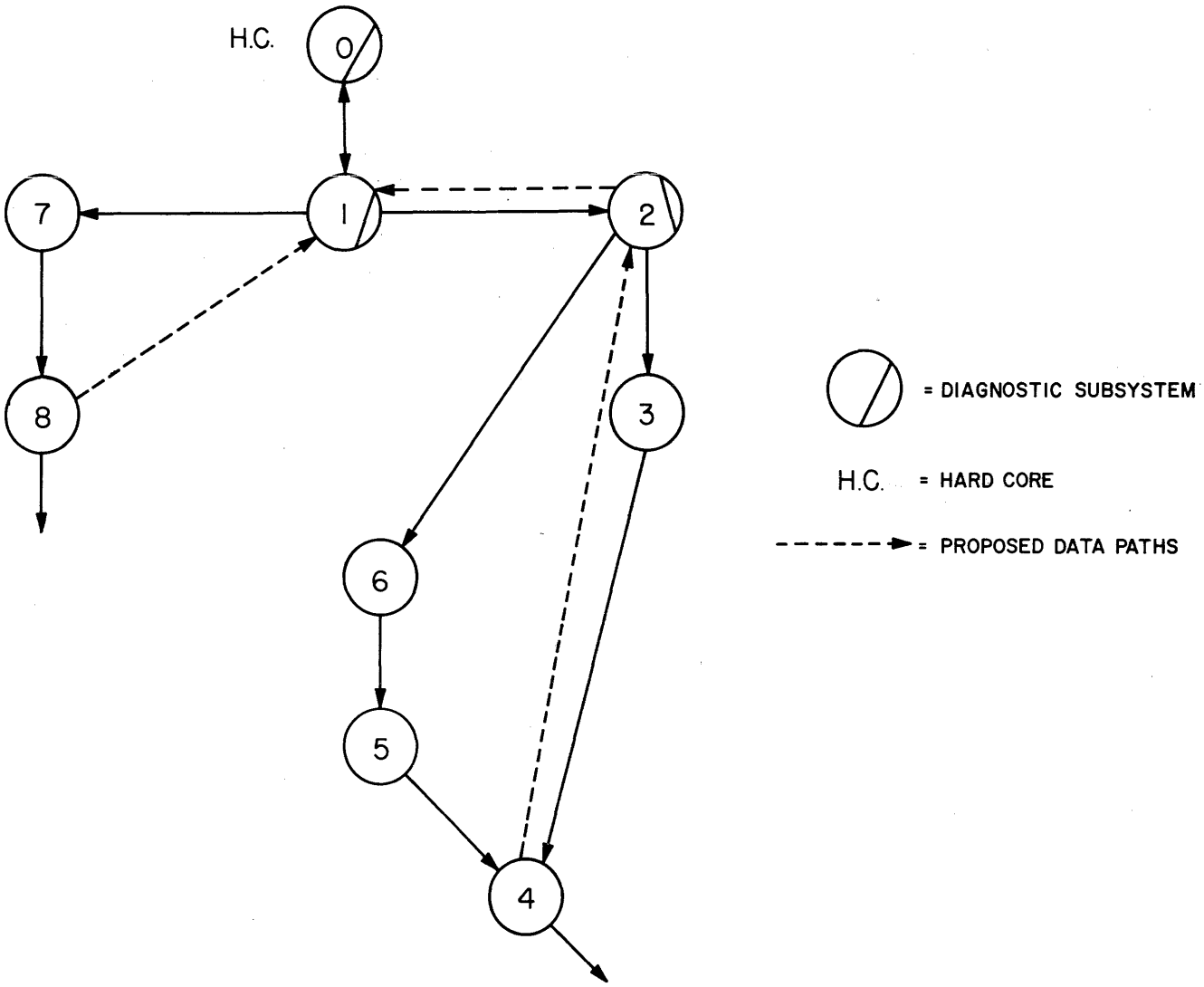


Figure 10—Achieving chain diagnosability

CONCLUSIONS

We have shown that the representation of discrete systems by graphs and their subsequent analysis is a valuable tool in system diagnosis. In particular, we have shown that one can use the theory of graphs in partitioning a system into smaller subsystems, in determining the strategic locations for test points, and in finding the sequences in which the system elements must be tested. We also have derived explicit methods for fault detection and fault location. Also, some consideration is given to the problem of self-diagnosability. It is important to remember that all these techniques depend only on the structural aspects of the system. When coupled with the behavioral characterizations of the system elements, they complete the parameters necessary to design diagnostic tests.

Partitioning a large system into its structural segments helps in the specification and assignment of components in module (major boards) and micromodules (flat-packs) (Figure 1). Also, the graph con-

nectivity consideration detects and locates logical errors in the information flow.[7] Similar techniques can be used in the debugging and maintenance of computer programs.

Much fruitful work remains to be done. Fast computer algorithms are needed to manipulate large graphs. Perhaps one can specify various degrees of connectivity. The basic methods of activating test inputs and monitoring the test outputs must be investigated.

ACKNOWLEDGMENT

The author acknowledges his indebtedness to his many colleagues at Honeywell for suggestions and discussions. He is grateful to Dr. R. B. Lawrance for his continued encouragement.

APPENDIX I—A modified method of selecting prime Implicants

1. Given the test point range matrix M, compute the row weights (number of ones in a row) and the column weights of the matrix.
2. Find the row with the least weight. Select the column with the largest weight which is implied by this row. Delete all rows subsumed by this implicant. Repeat Step 2 until no rows remain.
3. The selected columns correspond to the test points that are needed for almost minimal number of test point pairs.

For many simple cases, this algorithm provides optimal test point pair allocation. It is possible that variations of the above selection procedure can be found for improving the number of computations or closeness to optimality.

REFERENCES

1 R. ELDRED
Test routines based on symbolic logic statements
Journal of ACM Jan. 1959

2 W. KAUTZ
Automatic fault detection in combinational switching networks
Proc. Symposium of Switching Theory and Logic Design 1961

3 J. P. ROTH, et al.
Techniques for the diagnosis of switching circuit failures
Trans. IEEE Communications and Electronics Sept. 1964

4 W. C. CARTER, et al.
Design of serviceability features for IBM system 360
IBM Journal of Research & Development April 1964

5 S. SESHU, D. FREEMAN
Diagnosis of asynchronous sequential switching systems
IRE Trans.-EC Aug. 1962

6 R. E. FORBES, et al.
A self-diagnosable computer
Proc. Fall Joint Computer Confernece 1965

7 C. V. RAMAMOORTHY
Analysis of graphs by connectivity considerations
Journal ACM April 1966

8 C. V. RAMAMOORTHY
The analytic design of a dynamic look-ahead and program segmentation system for multi-programmed computers
Proc. National Conference of Assoc. Comp. Machinery 1966

9 C. V. RAMAMOORTHY
Generating functions of abstract graphs with systems applications
Ph.D. Thesis Harvard University May 1964

10 E. J. McCLUSKEY
Minimization of Boolean functions
B.S.J.T. November 1956

11 R. A. JOHNSON
An information theory approach to diagnosis
Proc. Sixth National Symposium on Reliability and Quality Control Jan. 1960
References 12 and 13 are excellent summaries on related topics.

12 R. WARD and T. O. HOLTEY
The maintainability factor in the design of digital systems using microelectronics
Proc. WESCON 1966

13 F. HARARY, et al.
Structural models
John Wiley & Sons New York 1965

Compiler level simulation of edge sensitive flip-flops

by JAMES T. CAIN and MARLIN H. MICKLE
University of Pittsburgh
Pittsburgh, Pennsylvania
and
LAWRENCE P. MCNAMEE
University of California
Los Angeles, California

INTRODUCTION

Recently there has been increased activity in the area of compiler level language simulation of digital systems. The compiler level language has a distinct advantage in that it places a valuable tool in the hands of the design engineer who usually has at least a prima facie knowledge of such a language.

R. P. Larsen, M. M. Mano,[1] and R. M. McClure[2] have recognized the need for some sort of low cost digital simulator to be available to all digital design engineers. They state that trends indicate more and more engineers are being confronted with digital design problems which are amenable to simulation techniques. However, many of these engineers are unable to use the powerful tool of simulation simply because the small size of the systems, or the fact that they are one or two-of-a-kind systems, does not justify the expense of developing a simulation program.

McClure and other individuals at Texas Instruments Inc. have attempted to fill the need for a low cost digital simulator by developing a special programming language for simulating digital systems.

Larsen and Mano have proposed using the FORTRAN IV standard programming language as a digital system simulation medium. The FORTRAN IV language was chosen because the logical Boolean operations are included in the instruction repertoire. They state that the simulation programming can be simplified through a systematic approach. In addition the digital network should be reduced to a set of the simplest possible digital elements (ANDS, ORS, NOTS, FLIP-FLOPS, etc.) and that the elements with memory (FLIP-FLOPS) should be simulated by a Boolean statement relating the element's new output in terms of the present input and output values. For example, the mapping relationship that they suggest for a trigger flip-flop is:

$$f(t_2) = s(t_1) + f(t_1)\, t(t_1) + r(t_1)\, t(t_1)\, f(t_1) \qquad (1)$$

where f is the assertion state of the flip-flop, s, the set signal, r, the reset signal, and t, the trigger signal. The letters in parentheses, t_1 and t_2, refer to the present and new states respectively. This is a mapping relationship not an expression of the internal hardware configuration of the trigger flip-flop.

An inspection of the above relationship reveals that the mapping is level sensitive, i.e., it corresponds to a Boolean function. In simulation, a flip-flop is used considerably as a module in the building of ring counters, binary counters, etc. The counters which are built from these flip-flops depends on the edge-sensitive properties of the modules. In order to provide the simulation of devices such as a ring counter, it will be necessary to arrive at a mapping "relationship" in terms of a compiler level statement or statements which demonstrate the edge-sensitive properties of the flip-flop module. The purpose of this paper is to provide a compiler level subroutine which in itself is edge-sensitive. For the purpose of demonstration, the ring counter of Figure 1 will be used as a vehicle.

Simulation

The MAD (Michigan Algorithmic Decoder) language was chosen because (1) the system can compile and execute programs in the Boolean mode (i.e., Boolean equations can be written and input directly to the compiler), (2) the MAD language, like FORTRAN IV, can easily be available to digital design engineers, and (3) the language has powerful simplified input and output statements which augment the power of the language as a digital simulator.

A close inspection of the ring counter circuit of Figure 1 reveals that the S-R flip-flops used in its construction must be sensitive to the trailing edge of the set or reset signal. In simulating the flip-flops needed for this ring counter, two problems are encountered. First, a flip-flop is a memory element which means that feedback of signals is incorporated in its

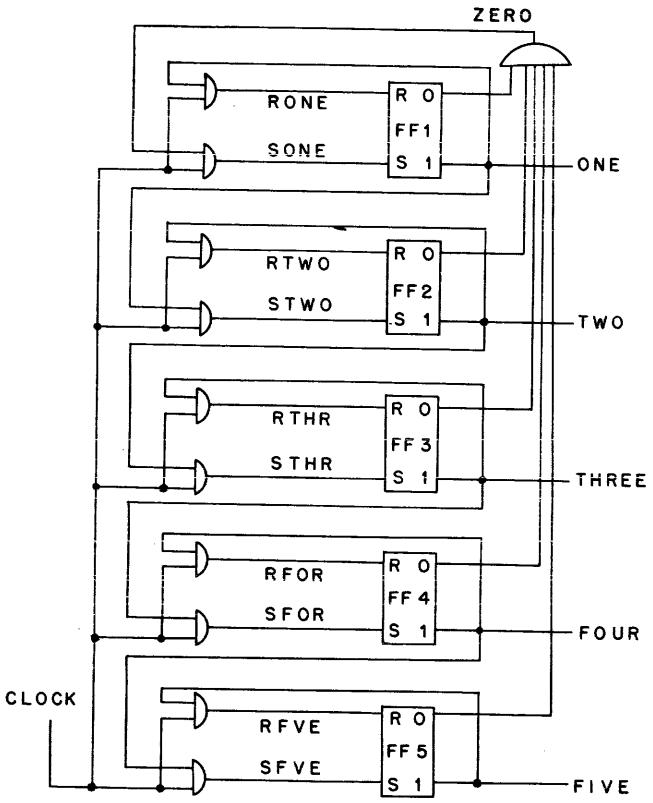


Figure 1—Ring counter

operation. This feedback action in an actual flip-flop takes place in the parallel mode, but in the simulator, all operations must occur in the serial mode. Second, the set and reset terminals of the actual flip-flops are sensitive to the trailing edge of the signals so that this type of operation must be included in the simulation program.

The simulation of the trailing edge sensitive flip-flops in the ring counter was accomplished by using the external function capability of the MAD language. An external function is actually a subroutine which is compiled independently of the main MAD program. The external function for trailing edge sensitive flip-flops is shown in the following external function. The statements are numbered in the left hand margin for reference purposes.

EXTERNAL FUNCTIONS FOR TRAILING EDGE SENSITIVE S-R FLIP-FLOPS

```
1        EXTERNAL FUNCTION (S,R,A)
2        NORMAL MODE IS BOOLEAN
3        INTEGER I, II, IN, J, K, L
4        INTEGER FAKO1
5        DIMENSION SS (9), RR(9)
6        PROGRAM COMMON FAKO1, FAKO2
7        ENTRY TO TFFSR.
8        FAKO1 = FAKO1 + 1
9        WHENEVER FAKO1.E.1
10       K = 5
11       L = 2*K-1
12       II = K-1
13       J = II
14       THROUGH INIT2,FOR IN=0,1,IN
15       .G.II SS (IN) = OB
16       RR (IN) = OB
17   INIT2 CONTINUE
18       OTHERWISE
19       CONTINUE
20       END OF CONDITIONAL
21       WHENEVER J .E. L
22       J = K
23       THRODGH TR, FOR I = 0, 1,I .G.II
24       SS(I) = SS(I+K)
25       RR(I) = RR(I+K)
26   TR  CONTINUE
27       OTHERWISE
28       J = J + 1
29       END OF CONDITIONAL
30       SS(J) = S
31       RR(J) = R
32       WHENEVER .NOT.SS(J).AND.SS(J-K)
33       SSS = 1B
34       OTHERWISE
35       SSS = OB
36       END OF CONDITIONAL
37       WHENEVER .NOT.RR(J).AND.RR(J-K)
38       RRR = 1B
39       OTHERWISE
40       RRR = OB
41       END OF CONDITIONAL
42       FUNCTION RETURN SSS
         .OR.(.NOT.RRR.AND.A)
43       END OF FUNCTION
```

Statements 1 through 6 are defining control statements. The variables S, R, and A, in statement 1 are dummy variables. When the subroutine is called, the actual program variables, which are in corresponding positions in the calling statement, are carried into the subroutine via these dummy variables. For this particular subroutine the dummy variable S is the set signal, R is the reset signal, and A is the output signal of the assertion side of the flip-flop. Statements 2 through 6 provide information for the compiler. These statements are necessary because the external function is treated as a separate program by the compiler.

The actual entry point to the subroutine, when it is called by the main program, is statement 7. Statements 8 and 9 are control statements which provide a means of initializing the subroutine when it is called for the first time. Statements 10 through 20 are the initialization statements. Statement 10 indicates to the system how many flip-flops (K=number of flip-flops) are to be simulated and the remaining initialization statements are a function of this statement.

The portion of the subroutine between statements 21 and 29 is designed to save core space. Since five flip-flops are being simulated, only five set and reset

signals need be saved. This enables the previous signal to be compared with the new signal to determine if it is the trailing edge of the signal. After all five flip-flops have been evaluated, these statements save the last five set and reset signals in SS (0) . . . SS (4) and RR (0) . . . RR (4)

Statements 30 and 31 transfer the current value of the set and reset signals, carried by the dummy variables, to the linear array. Statements 32 through 36 compare the current value of the set signal with the previous value for that particular flip-flop. If the signal has changed from high to ground (1 to 0), the trailing edge of the pulse is present. Therefore statement 33 sets the effective set signal to one. Otherwise the effective set signal is set to zero. Statements 37 through 41 perform the same function for the reset signals. After the program has evaluated the effective set and reset signals, the new value of the assertion terminal of the flip-flop can be evaluated by a mapping relationship between the input and previous output signals. Statement 42 evaluates this mapping relationship and returns to the calling program with the new value of the flip-flop output. Statement 43 serves only to tell the compiler where the external function definition terminates.

The following MAD program demonstrates how the external function is used in the simulation of the ring counter.

```
RING COUNTER
THROUGH END, FOR I = 1, 1, I.G.50
CLOCK = .NOT .CLOCK
    SONE = CLOCK .AND.ZERO
    RONE = CLOCK .AND.ONE
ONE = TFFSR. (SONE, RONE, ONE)
    STWO = CLOCK .AND. ONE
    RTWO = CLOCK .AND. TWO
TWO = TFFSR. (STWO, RTWO, TWO)
    STHR = CLOCK .AND. TWO
    RTHR = CLOCK .AND. THREE
THREE = TFFSR. (STHR, RTHR, THREE)
    SFOR = CLOCK .ANR. THREE
    RFOR = CLOCK .AND. FOUR
FOUR = TFFSR. CSFOR, RFOR, FOUR)
    SFVE = CLOCK .AND. FOUR
    RFVE = CLOCK .AND. FIVE
FIVE = TFFSR. (SFVE, RFVE, FIVE)
END ZERO = .NOT.ONE.AND..NOT.TWO.AND
    .NOT.THREE.AND..NOT.FOUR.AND..
    .NOT.FIVE
```

It is important to note that, by using this subroutine, statements were written which have a one to one correspondence to the actual hardware configuration of Figure 1. The group of statements which simulate the individual flip-flops must be ordered as shown because the set and reset signal of the next flip-flop depend upon the state of the previous flip-flop. In order to demonstrate the action in the ring counter, a MAD through loop was used. By the use of a through loop, an input clock signal is generated by a single statement, i.e.,

```
CLOCK = .NOT. CLOCK
```

The external function as presented is sensitive to the trailing edge of the incoming waveform. The function can be easily modified in order to sense the leading edge. The modification is performed simply by rewriting statements 32 and 37 of the external function as shown below.

```
32  WHENEVER .NOT.SS(J-K).AND.SS(J)
37  WHENEVER .NOT.RR(J-K).AND.RR(J)
```

SUMMARY

The authors believe the contribution of this paper is the development of the external functions or subroutines to simulate the edge sensitive devices which are an integral part of a digital system. The development of this and other subroutines enhances the power of the MAD language for use by non-programming oriented, digital design engineers. This is because the use of the subroutines produces a one-to-one correspondence between every logic gate in the simulated system and every MAD logic statement or subroutine call in the simulation program. This enables the design engineer to use the logic schematics of the computer system as flow charts for writing his simulation program.

The subroutine was constructed to be as flexible as possible. The external function capability of the MAD language was used instead of the internal function capability because the Michigan system treats an external function as a completely separate program. This means that a variable in an external function can have the same name as a variable in the main program or another external function and no multiple definition problems will result. This feature augments the flexibility of the subroutine which was developed in this paper.

The subroutines presented can easily be adapted to simulate any number of trailing edge sensitive flip-flops. A similar subroutine has been developed to simulate leading and trailing edge sensitive trigger flip-flops.

REFERENCES

1 R P LARSEN MM MANO
Modeling and simulation of digital networks
Communications of the ACM 8 308-312 (May 1965)

2 R M McCLURE
A programming language for simulating digital systems
Journal of the Association for Computing Machinery 12 14-22 (January 1965).

A logic oriented diagnostic program

by HERMAN JACOBOWITZ
Radio Corporation of America
Camden, New Jersey

INTRODUCTION

The automatic diagnostic program has been with us for almost as long a time as the electronic digital computer, however the disparity between its achievements and its potentials has not been as successfully closed as it has for the computer itself. In the opinion of the author this is at least partly due to an only recently acquired understanding that the term "diagnostic program" is a misnomer insofar as it implies that the task is primarily one of programming rather than logic analysis.

For this reason, during the development of a test and diagnostic system (T&D) for the MICRORAC FIELDATA COMPUTER SYSTEM (RAC), a series of studies was undertaken to clarify both the performance attainable by a diagnostic program and the methods for achieving this performance. The results of these studies are summarized in the following sections.

The quality of a diagnostic program

The detection capability

This relates to the ability of the program to determine that a fault exists in the system. This may be integrated with the portion of the system that produces an actual fault diagnosis, however, it is desirable that the detection system be capable of rapid, easy application, since the system must be tested frequently enough to ensure that only a single fault exists at any one time. It is obviously important that the percentage of undetected faults be kept as small as possible, however this system has not been specifically designed to detect intermittents, marginal faults or faults in redundant logic. For example, in some cases, logic elements are gated by a particular timing pulse in order to ensure correct operation under worst case conditions. Since the worst case almost by definition, rarely occurs, it is clear that a failure which causes certain logic elements to be clocked at an improper time will generally not be detected. Similar conditions occur in certain logic areas used in the core memory system to minimize noise disturbances. Marginal testing techniques must be relied upon to deal with such faults.

The diagnostic resolution

In general, most diagnostic procedures will lead the operator to the conclusion that one of a small group of possible faults exist. Although it is frequently possible to refine the procedure by additional automatic semi-automatic or manual tests, to lead to a unique diagnosis, this is not always economically desirable because the RAC maintenance system includes the use of an automatic card tester capable of thoroughly testing every module on a card. The use of the tester, however, requires a fairly long time per card and also requires that cards be removed and reinserted into the RAC frame. This latter procedure, if repeated too frequently puts additional undesirable stresses on the connector and may increase its failure rate. For these reasons the resolution or number of possible faulty cards listed at each diagnosis varies from point to point. The final resolution reflects a compromise between the cost of developing additional diagnostic procedures and the desirability of minimizing unproductive usage of the automatic card tester.

The required operator skill level

The skill level required to maintain a digital computer could conceivably vary all the way from graduate design engineers to laymen completely unfamiliar with electronic equipment, but capable of following detailed explicit directions. Unfortunately many military systems have been built with the former requirement while few have approached the latter.

If a completely automatic system were developed to diagnose 100% of all faults to a single card, a degree of training would still be required because the present nature of computers requires tape handling, program loading, initialization and starting, card replacement, etc. These skills are essentially at the level of the computer operator rather than the maintenance man.

The time required for diagnosis

This should be distinguished from the total repair time and from the time required to extract and handle cards and modules because these latter times are more a result of computer design and operator training than diagnostic system design. For this reason diagnostic time is regarded as beginning with a loaded test and diagnostic program and ending when the operator possesses a list of fault cards. In considering a reasonable time requirement it is obviously misleading to call for a time disproportionate with the other time elements in the repair and maintenance cycle, i.e., a diagnostic time on the computer time scale of microseconds or milliseconds is useless because the operator activities required to complete the diagnostic cycle occur on the human scale of seconds or more likely minutes. On the other hand, some elements of the diagnostic test procedure will undoubtedly be used every day and so must take only a modest portion of allowed maintenance time. This suggests that a reasonable time might vary from 1 minute to 1 hour depending upon whether a quick reassurance of system operability is desired, to diagnosis of a subtle fault requiring manual intervention and analysis. In this light a goal of 10 minutes average diagnostic time has been used.

The additional equipment required to reach a diagnosis

Because the state of the art of computer design has not advanced to the point wherein a computer can be produced which is completely and automatically self-diagnosable, it is clear that provisions must be made for operator intervention to perform semi-automatic and manual tests. Such tests may require a wide range of equipment from a simple D.C. voltmeter to an additional test computer. The test philosophy adopted was to provide for a range of techniques and levels of manual intervention, largely at the discretion of the operator. Thus each diagnosis results not only in a list of cards but a set of references which usually indicate to the operator the following additional information:

1. The group of logic elements and gating signals being tested and their normal functions.
2. The normal results of the tests being performed.
3. The tests previously performed and functions and elements found to be correctly operating.
4. Logic drawings references.
5. Pin number and test point number identification data.

This information will frequently enable an operator to refine, verify or extend a diagnosis by simply measuring the D.C. potential or observing the presence or absence of a pulse at a particular test point.

The criteria for automatic diagnosis

To execute *any* automatic diagnostic program, it appears necessary that a fault not affect the ability of the computer to perform the following functions:

1. Load a program into memory.
2. Automatically sequence through the instructions of the program.
3. Automatically perform a comparison of a result with a known correct result.
4. Automatically execute a conditional transfer of control, as a result of a comparison.
5. Automatically print or display a result.

If a fault makes it impossible to perform any one of these five functions, it is not true that diagnosis is impossible but, rather that such a failure requires manual intervention for its interpretation and so contributes to an excessively long or complex diagnostic cycle.

It is quite difficult to apply these criteria to obtain precise numerical estimates because each of these functions can usually be performed in many ways, even in a computer not particularly designed with diagnostics in mind, much less a military computer such as RAC. For example, if program loading via magnetic tape is impossible due to element failure in one of the I/O converters, a few minutes of cable switching allows the magnetic tape to be loaded via a second I/O converter, or the magnetic tape may be replaced by a paper tape entry using the same converter. If the fault persists, short programs can be loaded, one instruction at a time via the console.

It is thus clear that there are many possible roads, albeit manually oriented, to the initiation of a final automatic diagnostic program. Therefore an estimate of the amount of hardware which must be operative to perform an automatic diagnostic program, in some sense of the word automatic, is largely limited at the top by the ingenuity of the program designer as well as the characteristics of the computer design. However, the problem of economic feasibility limits the number of possibilities which may be explored to provide alternate automatic or semi-automatic paths for automatic diagnostic execution. For this reason the test and diagnostic system for RAC makes no claim to exhaustive use of all possible as well as probable alternatives. However, among the redundancies included in the system are:

1. Program entry by any one of two I/O converters.
2. Program entry by either paper tape or magnetic tape.
3. The use of alternate core memory banks.
4. Results displayed either by a High Speed Printer on IOCNV# or a Flexowriter on IOCNV #2 or from console indicators directly on the B Register

5. The use of logical instructions to synthesize a redundant comparison mode.
6. The use of a single instruction, console controlled, diagnostic mode for use in diagnosing failures which otherwise cripple the ability of the system to "breathe."

The relationship of card partitioning to diagnostic resolution

It has been observed that the placement of functionally related logic on the same or a small number of circuit cards makes it unnecessary to diagnose or analyze failures down to the logic element level. This occurs because diagnosis or isolation to a function then allows one to replace the card or cards concerned with performance of this function. Two objections exist to this viewpoint:

1. Because of a requirement for the MICRORAC system to minimize the number of card types and resulting inventory costs, RAC uses only 30 different board types. It was therefore not possible, in most cases, to place only functional related logic on a board. Thus, to determine that a circuit board is faulty, it is generally necessary to determine that a specific logic element on the board is faulty.
2. Even if the circuit boards did contain only functionally related logic, the problem of diagnostic resolution would only be partially ameliorated. This occurs because faults which are not functionally related to a particular test, and therefore, could occur on any card, may still affect a particular test.

A more detailed analysis of logic failures indicates that the ability to use partitioning as an aid to diagnostic analysis is most significantly affected by the presence of two types of faults which may be categorized as:

Type 1 Faults: Those which prevent an action from occurring or:

Type 2 Faults: Those which cause an action to occur, improperly. Localization of Type 1 faults can be made easier by appropriate partitioning, because such faults are those which are ordinarily considered as being related to the function under consideration. However, localization of Type 2 faults is not aided by partitioning because these faults can originate any place in the computer.

It thus appears that the problem is not one of diagnosing to 1, 2 or 3 boards, but rather the diagnosis to 1, 2 or 3 elements, each of which may be on different boards. The magnitude of the latter problem may be demonstrated by observing that the Central Processor of MICRORAC contains about 160 cards. Diagnosis directly to 3 cards, would imply a resolution of approximately 2%. However, the number of elements in the Central Processor is about 5000 and diagnosis to 3 elements implies a resolution of 0.06%.

The effective number of logical elements in a diagnostic system

It has been demonstrated that the analysis for a diagnostic program cannot be content with a card by card analysis i.e., that essentially every logical element must be considered. However, an even deeper level of analytical detail is required i.e., every failure mode of an element must be analyzed in order to develop a thorough diagnostic program. This can be seen by considering the basic methods of diagnosis. All methods involve an element of prediction. The prediction process consists of analyzing the effects of a presupposed failure upon observable portions of a computer. This is essentially a method for building up a conceptual dictionary relating failures to symptoms. Obviously, every entry in the table results from selecting a logic element, assuming its failure and then deducing its symptoms. Each element can, however, fail in several ways, each of which can produce a characteristically different symptom. For example, consider a two input - non - inverting AND gate, with output Z and inputs A and B. The modes of operation of this gate including normal operation and all types on non-marginal faults are:

$$Z=AB \quad (1)$$
$$Z=A \quad (2)$$
$$Z=B \quad (3)$$
$$Z=0 \quad (4)$$
$$Z=1 \quad (5)$$

In addition, two other modes exist in which each of the input diodes is shorted. Such an element failure does not alter the transfer function of the gate, however, it has the disturbing effect of altering the operation of gates sharing the same input leads, thus producing a sidewise sneak path.

In any case, it is clear that the prediction method requires the consideration and analysis of the equivalent failure mode of operation of every gate. In effect, if the average gate of flip-flop has six modes of failed operation, each normal logic element is replaced, sequentially, by six other logic elements. The effects of these new logic elements must be deduced, thus multiplying the required extent of analysis by six.

CONCLUSIONS

As an example of the successful application of these principles, the RAC Diagnostic System, consisting of approximately 22,000 instructions and data words, and over 7500 individual tests leading to specific diagnostic listings was subject to a two level acceptance test. 400 failures were inserted by RCA during test and debugging of the system and an additional 75 were inserted during government conducted acceptance tests. 100% of all faults were detected, 74% by explicit module listings, to a median resolution of 4 cards out of more than 500 cards. The median diagnostic time was 2 minutes. Implicit diagnosis of unit and function was provided for the remaining 26%.

ACKNOWLEDGMENTS

The author wishes to record the outstanding efforts of a few members of the group who contributed so heavily to the success of this program from conception to final test. They are H. Hellman and M. Kornfeld of the Simulation, Automation and Diagnostics Group. The programming efforts were directed by A. Rose, who performed masterfully in bridging the gap between logic analysis and programming. Assistance in significant areas was provided by P. Carides, J. DeSantis, J. Cohen, B. Burghen, M. Sendrow.

Automatic trouble isolation in duplex central controls employing matching

by E. M. PRELL
Bell Telephone Laboratories, Incorporated
Holmdel, New Jersey

INTRODUCTION

At present in stored program control systems for switching telephone traffic, duplication is provided for dependability rather than to increase the traffic handling capacity of the system. Figure 1 depicts a simplified block diagram of the data processor portion of such a system.* The data processor is composed of central control and memory subsystems interconnected by communication buses. The memory subsystem** comprises two store groups each containing numerous information blocks (or stores). Each store of a particular store group has its corresponding image in the opposite store group. The stores contain two distinct categories of information: (1) semipermanent data (program and parameter), and (2) traffic variable data which are sometimes intermixed, but generally separated into different stores or even different memory subsystems. The data processor can be thought of as two central controls (CCs) which normally run in parallel, synchronously executing the same program, each from its own associated store group.

To be able to perform useful work in the switching system environment and for the system to be able to maintain itself, the CCs must also be able to receive information (scan) from points not normally accessible within their own environment and send information (distribute) to units outside of their own environment. One CC is defined to be active and the other standby. Normally only the active CC controls the I/O equipments.

The data processor must be able to control I/O equipments which operate at speeds incompatible with its basic processing speed. This requires that the system be multiprogrammed. Also, it is more efficient in a multiprogrammed system to employ interrupts to allow each program subsystem to operate at a rate appropriate for the equipment and/or processing function involved. For example, one interrupt level allows I/O programs to be executed at rates dictated by the I/O equipments. Higher interrupt levels allow maintenance programs to make high priority claims upon the data processor when trouble occurs.

Duplication and switching are necessary to allow the system to operate in the presence of hardware troubles. Thus, for example, the system must be able to operate with a store out in store group 0 and a central control out in data processor 1, see Figure 1. Generally, all communication buses are completely switchable at the central control; i.e., either or both buses can be connected to either or both central controls. Hence, various central control-communication bus modes can be established by means of unique configurations of the buses and central controls.

These systems depend primarily on hardware checking circuits for trouble detection during data processing. When a trouble detection circuit detects a trouble in the system, it notifies an "interrupt" circuit. The interrupt circuit immediately stops operational program processing and transfers control to a fault recognition (FOR) program associated with the particular type of trouble indication. The functions of the FOR programs are to determine quickly an

*For a more complete description the reader is referred to Reference 1.

**For simplicity, a store-communication bus system where a store is split into two functional parts and where each half can be connected to either or both communication buses (called split-duplication) is not shown. In this method, the input and output buses to and from the stores are logically separate entities with independent bus control; e.g., a store half could input from bus 1 and transmit to bus 0.

operational system configuration, establish it by switching out faulty units, and then return to operational program processing. Sometimes, the ability of these programs to perform their function is impaired by the fault in the system. When this occurs, a hardware back-up system in conjunction with a checking program, control the data processor reconfiguration. This facility is designed as an autonomous circuit within the central controls and is initiated by various timing circuits.

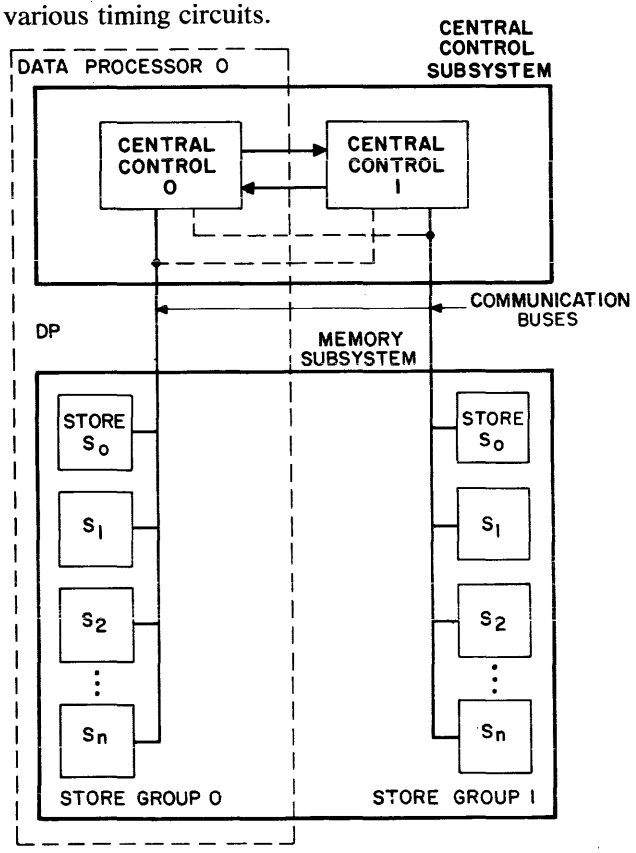


Figure 1 – The data processor

Thus, the FOR programs are basically tests of processing integrity (sanity) of the system. These programs, with the help of additional hardware features, minimize the probability of an inoperative configuration mutilating temporary information. To minimize the amount of time taken from operational program processing, these programs generally try to use brief and direct methods to discriminate between error conditions (non-repeatable) and repeatable troubles (faults). When a fault is detected, sufficient testing of the duplicate units is performed to find one complete working system, and after doing so, to return to operational program processing.

Fault Recognition is the most important maintenance function in a real-time system. The minimization of time taken from the processing function with an extremely high probability of returning to processing with an operational (sane) system is the critical objective which any automated maintenance technique must satisfy. Diagnostics with "fine" resolution are of secondary importance.

The FOR programs will request that each unit found faulty or suspected of being faulty be diagnosed. The function of the diagnostic programs (which are interleaved with operational program processing) is to generate test data to isolate the fault to a reasonably small section within the unit. Maintenance dictionaries are provided for translating these test results to the location of the faulty section. By using standardized packages and plug-in techniques, faulty components can be readily removed from the system.

The central control is the most difficult part of the system for which to design maintenance programs. This is because of its critical position in the structure, its size, and its lack of a high degree of symmetry. This paper concerns itself with two basic methods developed at Bell Telephone Laboratories to cope with this problem.

Central control maintenance – general considerations

During operational program processing as mentioned earlier, the central controls are normally synchronously executing the same program. In addition, a circuit between* the central controls is used to inform each central control of the operations being performed by its mate. This circuit is called a match circuit. It provides programmatic access to the microsteps within the complex central control operations. A simple way of looking at the match circuits is as follows: central controls 0 and 1 add $1 + 1 = 2$. Then each central control with its own** match circuit verifies that the same inputs and outputs occurred with the adders of the two central controls. When either central control detects a mismatch, a FOR program is entered.

The first purpose of FOR programs is to filter out random errors. If the match circuits are fast enough, and the machine language structure clean enough, the FOR programs can actually unwind the instruction (or instructions) so that a retrial of the failed operation can be attempted. Unwinding generally refers to reestablishing the state at the beginning of a partially executed instruction or of a nested sequence. A second failure usually implies a faulty unit. Otherwise, an error occurred and both machines are returned to the interrupted operational program.

However, in some machines the logical complexity is such that they cannot be stopped on mismatches

*Functionally only.

**In some machines one central control may look at inputs at one moment in time and the other at outputs.

before program accessible data in the machine has been mutilated. In this case, program unwinding is impossible. Hence, what the matcher (or matchers) match then must be a function of the operation (or operations) in progress within the central control(s). In this case, the FOR program actually attempts to reproduce the trouble symptom under controlled conditions by logically exercising the central control hardware which most likely caused the interrupt. (This exercise program is a subset of the complete FOR program.) If no trouble symptom can be reproduced, the problem is classified as an error for subsequent error analysis† and the machines return to operational program processing. In either case these programs lead to a complete check of all central control hardware if a fault symptom is detected.

There are two basic techniques for fault localization. The first method makes use of a decision tree where at each node within the tree, the fault space is partitioned into two or more hopefully disjoint and almost equal smaller possible fault spaces. Resultingly, the tree leads to a set of terminal nodes each of which determines a few possible faults in the unit. Because of the difficulty of designing a decision tree for relatively large complex circuits such as a central control, a method is sought based on executing a more or less fixed set of functional tests which produce an output derived from the failing tests. This output would then be program translated into a number which would identify the faulty circuit pack(s) in a printed dictionary which would be compiled by fault simulation.[2] This paper considers diagnostic test design basically from the latter point of view. In addition, the difficulties which arise in attempting to consider the entire unit under diagnosis at the time of test design led naturally to the concept of dividing the unit into functional circuit blocks, and then considering each block individually. In doing this, the program designer, when testing block i, assumes that the fault is in block i. Thus, the other circuit blocks can be used as test tools in testing block i. But the division of the program into functional blocks of tests (called phases) creates relatively complex interfaces between these blocks as well as creating many problems relating to consistency[3] of the test results.

To minimize the "hard-core" of circuits which must function to properly execute the FOR program, much thought must go into the functional division and ordering of the functional blocks. The functional division and ordering is even more important in diagnostics due to inconsistency† problems.[3] The objective is to use only tested circuits to test other circuits.

†Excessive errors rates can lead to complete checks and diagnosis of both central controls. If the error rates do not subside, teletypewriter message(s) will follow indicating the inability of the programs to isolate the reason for the high error rates.

Distinct central control fault recognition and diagnostic programs

A. Introduction

In the No. 1 ESS[1] stored program switching system developed at Bell Telephone Laboratories, the FOR and diagnostic (DIAG) programs are separate entities. The basic strategies of the separate FOR and DIAG programs are outlined in this section. These strategies are largely carried over to the integrated program approach discussed in the next section. The differences in the two approaches lie primarily in program structure and the use of the matching facilities.

B. Fault recognition

The basic program flow technique used within test phases in FOR programs is shown on Figure 2. The tests are designed as if each central control is testing itself under the assumption of no redundant logic such that a single failure would go undetected. "Tests consist of data manipulation operations followed by conditional transfer orders which check for the proper circuit response. If the active central control fails, it will attempt to switch central controls (which it alone can do by program). The faulty unit, now standby, will be removed from service and the diagnostic program requested. If the active passes, checks are made to see if the standby is still in step (by examining the match circuits). If the standby is found to be out of step, it is removed from service, since it failed to follow the operations of the good active unit. This testing process is continued until all tests have passed or a faulty unit is found."[2]

C. Diagnostic

In the DIAG program, it is known that the standby is faulty and that the active is fault-free. Thus, the active can be used to test the standby. The assumptions of single failure and no redundant logic are applied when designing tests. Again, the test programs are broken into phases. The execution of a test follows a strict pattern as follows:

†By inconsistency it is meant that when a given fault is inserted during fault simulation, the diagnostic program does not produce the same set of failing tests with repeated diagnosis or that the set of failing tests produced during laboratory fault simulation is not identical to that produced when the same (as the simulated) fault occurs in the field.

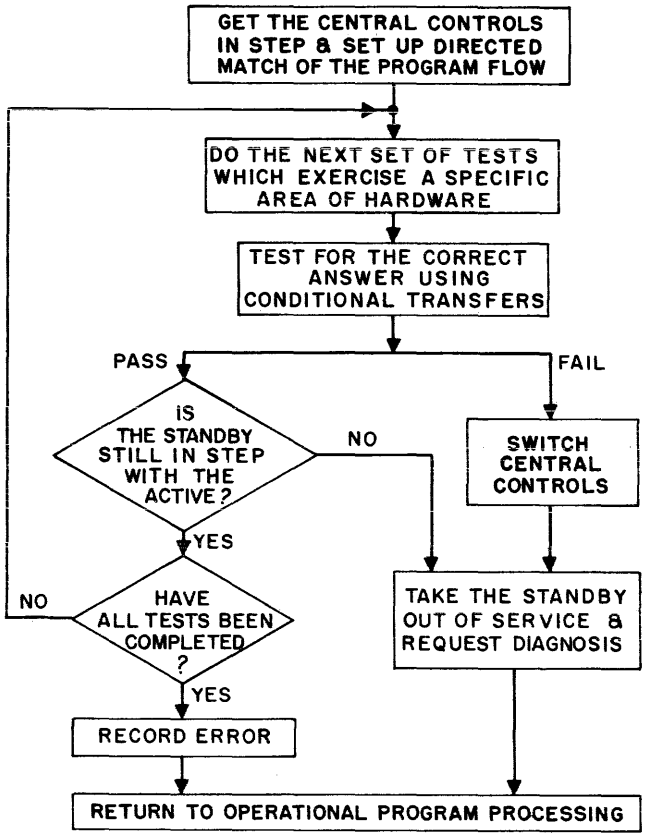


Figure 2—Central control fault recognition test mode

1 Initialize circuits external to the circuit being tested so their interaction on the testing circuit will be consistent,
2 Apply inputs to the circuit under test,
3 Observe the outputs,
4 When necessary, deactivate or reinitialize the tested circuit,
5 Compare outputs with expected outputs, or the active results, and
6 Record results indicating whether the test passed or failed, and if it failed, record results about how it failed.

In this method, the central controls are forced into step synchronously executing the same program. The initialization instructions are executed and then the active central control using the match circuits, takes a snapshot* of the execution of a test instruction. In addition to a mismatch indication, the "exclusive or" of the active and standby match results provides additional resolution within the test points being sampled.

*The match circuits can be used to selectively sample a specified point (or points) (having match access) a given number of machine cycles from the point of initialization and at a variable point (one of three time segments) within the desired cycle.

The integrated approach to central control fault recognition and diagnostics

A. Introduction

"Combined" and "Integrated" are used to define a single program which is used for both FOR and DIAG. "Combined" denotes a program which simultaneously performs FOR and DIAG. This is the method usually employed in commercial computers.[4,5] This approach cannot be used in systems which operate in real time because the execution time of a program which records diagnostic data from and through a match system is at least an order of magnitude greater than that of the corresponding program without data recording.** This type of increase in FOR execution time from say 25 MS to 250 MS cannot be tolerated in most real-time systems, and specifically not in the No. 1 ESS.

"Integrated" denotes a program whose course of action is dependent upon its use; i.e., FOR or DIAG. This method is based on the assumption that with sufficient match access, FOR is a subset of DIAG; that is, those procedures which test the logical capability of a central control also provide results for diagnostic resolution. For example, the logical tests used to test an adder during fault recognition would also be employed during diagnosis except that in diagnosis the matchers would be used to look at the input and output of the adder.

B. Matching features essential to this method

The primary means of trouble detection for the central controls is the matchers. They do, however, have a secondary function of providing a means for diagnosis of the central controls; i.e., they provide access to many points within the central controls. Since the match system does play a major role in the maintenance scheme, it should be intrinsically reliable. This means that the failure rate of the match circuitry should be small in comparison to the failure rate of the central control. To this end, it is imperative that the matchers be designed mainly for adequate trouble detection. Next, a judicious choice of the requirements for system diagnostic capability should be made if the matchers are not to be overly complex and become a burden on the system.

Basically the match system must be able to perform both a directed match and a sampled match. In a directed match, the matchers are directed to look

**Some of the reasons for this are that in DIAG the match sources and time segments (see section B) must be continuously changed to obtain the desired diagnostic data, whereas, in FOR the sources and time segment(s) are fixed throughout execution. In addition, in DIAG each failure or even each test requires data recording, matcher reinitialization, truncation decisions to avoid inconsistency, etc.

at match sources at specified time segments on a continuous basis.† In the sampled match, the matchers are directed to look at specified sources at a specified time segment a given number of machine cycles from the point of initialization. This is frequently called selective, discrete, or snapshot matching.

The No. 1 EES relied heavily on discrete sampling at predetermined points within a manually written test program. This is a powerful tool and it is used in certain DIAG only tests in the integrated program method.

However, the integrated tests use the directed match mode with flexibility provided by varying the match sources and/or the segments of the machine cycle when matching occurs. In this mode the specified sources are matched continuously on every machine cycle at one or more of the three time segments. This is the same mode that is used during operational program processing, but then the sources are set to the internal communication buses within the central controls. In diagnosis a special interrupt facility is initiated by the matcher(s) to call in a diagnostic recording routine when an abnormality†† is detected. In addition, the matcher will indicate the number of machine cycles or instructions since the last interrupt. This indication is referred to as a time-out count (TOC). The special interrupt facility can also be used with discrete sampling.

C. Integrated approach

In this approach a single functional block (or phase) of tests performs both diagnostic and fault recognition functions. The basic difference between using the program for FOR or DIAG is the collection of data. When this program is run as a FOR program, data about the standby central control is not collected.* Whereas, when the program is run as a diagnostic, a failure in test i will result in the saving of the TOC and the bit pattern of mismatch. A general flowchart for the integrated program for test i is shown on Figure 3. Note that test i now includes all instructions including initialization. Thus, this is more of a continuous sampling technique. In addition, the testing is continued regardless of failures and further data is gathered.

†A variation because of machine complexity is for the matcher(s) to be directed to look at a set of match sources on a sequential basis. The source to be matched may also become a function of the instruction(s) being processed.

††The matcher can be directed to interrupt upon detecting either a match or mismatch between the two machines.

*When executing the program for diagnosis, there exists the possibility of not executing phases that passed when the program was executed for fault recognition. This is accomplished if a minimum amount of recording is performed during fault recognition.

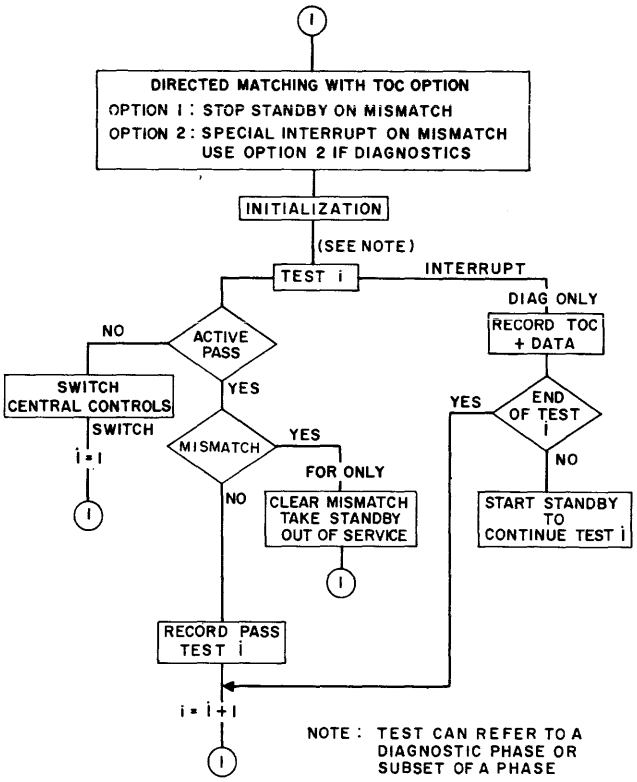


Figure 3 – Integrated technique

With this approach the part of the FOR program in the previously described approach used to completely test for faults is essentially eliminated. This results in approximately a 20 percent reduction in program size. However, in the integrated approach, there is a tendency towards some increase in recovery time for faults since more comprehensive diagnostic type test sequences tend to be used during fault recognition. It is also true that having separate programs allows more freedom in modifying the FOR program.

Further work

This section describes some of the more interesting ideas being considered for further improvements in these automated maintenance programs.

Two important factors in developing any automatic diagnostic technique are design time and memory cost. Both are large. Program sizes in terms of program words for diagnostics are approximately equal to the number of transistors in the unit. At present, the design of a set of sufficient tests has been obtained to a large extent by manual analysis and revised as indicated by empirical evaluation. Work has been going on for some time on automatic generation of a set of tests and the methods have been applied successfully to limited size circuit blocks. Prospects for applying these techniques to large combinational circuits look promising.

Memory costs will most likely be reduced in the future by storing diagnostic programs in a cheaper medium such as magnetic disk (or tape) and bringing them into temporary memory as required.

More work needs to be done in detecting and distinguishing different faults with a consistent pattern. Adequate distinguishability or resolution has not been too difficult to obtain, but detectability is only at about 80 percent of the set of faults considered. In designing a test for a particular class of faults, the reaction of this test to other possibly unrelated faults affects consistency. Possibly a combination of the decision-tree and failure pattern technique can bring about better consistency. The decision-tree would be used to partition the initial fault space into subsets of say a 100 or more possible faults and then the failure pattern technique could be used for further resolution.

Another possible avenue is the cell dictionary.[6,7] At present an exact match of the output number derived from the diagnostic results must be found in the dictionary of fault numbers generated by fault simulation. The cell dictionary would assign all faults to regions or cells of the diagnostic data space. If an exact match could not be found, the diagnostic cell dictionary output would list the closest cell centers to the actual diagnostic data pattern. This is only one of many other dictionary techniques being explored.[6,7] Early results look promising that one or more of these techniques will minimize the consistency problem.

Further improvements in mechanisms for data gathering to reduce the hard-core of circuits can be developed with an associated smaller matching system. The direct use of memory by the standby processor with the active processor sending store control signals looks promising here.

The present approach to self-diagnosis makes design changes difficult since the dictionaries usually must be regenerated. It would be highly desirable for small changes in circuit logic to be reflected in small changes in diagnostic programs and the associated dictionaries. Intermittent troubles are also a problem which are presently being solved by automatic error analysis followed by repeated diagnosis and off-line testing by highly trained maintenance personnel. How to function with less trained maintenance personnel who will have limited trouble-shooting experience due to the high reliability of electronic circuits remains a problem.

For the future, integrated circuits are going to require a whole new set of maintenance ground rules and revised automatic maintenance strategy and program. Hopefully, the work painstakingly done with the present type of circuits will make this task easier.

ACKNOWLEDGMENTS

The author is especially indebted to Messrs. R. W. Downing, W. C. Lehrman, and J. M. Lix for their valuable assistance during the development of the integrated program.

REFERENCES

1 R W DOWNING J S NOWAK L S TUOMENOKSA
No 1 ESS maintenance plan
Bell System Technical Journal part I September 1964

2 R W DOWNING J S NOWAK L S TUOMENOKSA
Maintenance planning on no 1 ESS
IEEE Annual Communications Convention Colorado June 1965

3 R L CAMPBELL
General maintenance techniques for large digital systems
5th Annual Reliability Maintainability Conference N Y C July 1966

4 K MALING E L ALLEN
A computer organization and programming system for automated maintenance
IEEE Transactions on Electronic Computers December 1963

5 R V BOCK A P TOTH
Hardware and software for maintenance in the B5500 Processor
IEEE International Convention N Y C March 1965

6 H Y CHANG
Methods of interpreting diagnostic data for locating faults in digital machines
IEEE Design Automation Workshop Michigan State U September 1966

7 H Y CHANG W THOMIS
Methods of interpreting diagnostic data for locating faults in digital machines
Bell System Technical Journal to be published

The place of digital backup in the direct digital control system

by J. M. LOMBARDO
The Foxboro Company
Foxboro, Massachusetts

INTRODUCTION

The key to the success of direct digital control on large industrial processes lies in its flexibility in implementing everyday process control problems as well as advanced control at lower overall system cost. Control concepts for continuous processes use the computing, monitoring, information storage and analytical ability of the direct digital control computer. In the batch or discontinuous process the computer's logic capability is emphasized. To perform batching operations, a comprehensive logic system is necessary. Implementation of such a system using digital techniques provides many advantages over implementation using analog equipment with auxiliary digital logic circuits.

To fully appreciate these advantages, the reader must have a basic understanding of continuous control systems as well as the batch type systems. The following will describe single loop control, several advanced control concepts and control of semicontinuous processes, as an introduction to digital computer application and backup.

Single loop control

Simple single loop feedback control is the most common control found in the process industries. It is used for controlling flow, level, temperature, pressure and many other variables. Both pneumatic and electronic devices are available which provide this type of control.

Basically, these controllers compare the measurement of a variable with its desired value or set point. If the two values are not equal, the controller adjusts a control value to minimize the difference (Figure 1).

In action, the controller is an analog computer which calculates a one, two or three term expression,

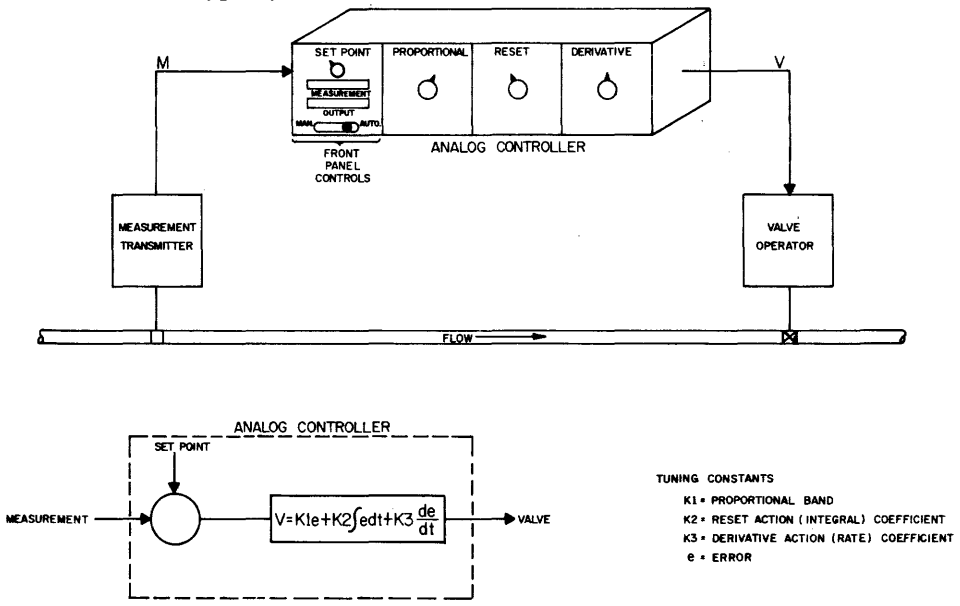


Figure 1 – Typical single variable feedback control loop

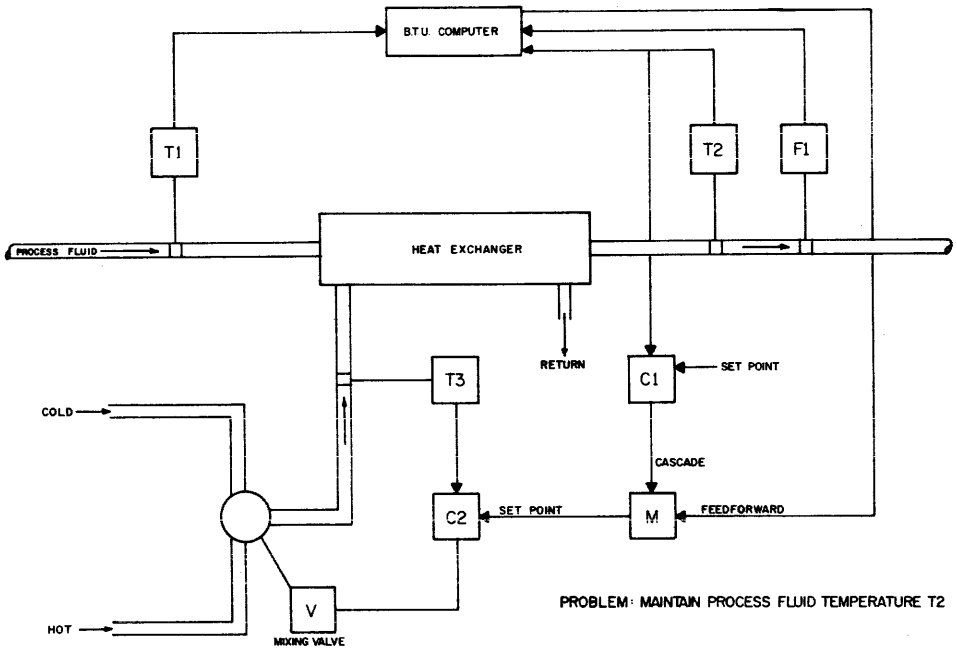


Figure 2—Advance control techniques applied to a heat exchanger

depending on the type of control action required by the process. The three terms define proportional, reset and derivative control action. During process start-up, coefficients of the three terms are manually set on the controller to provide the best response under normal operating conditions. If operating conditions change, or the process operator changes the set point radically, the coefficients are no longer at optimum values.

Advanced control concepts

As the control problem becomes more complicated, single loop feedback control is no longer sufficient. Figure 2 illustrates three types of advanced control: inferential, feedforward and cascade.

In the inferential control, a relationship is calculated between two or more measurements which is used to control the desired but unmeasurable variable. In Figure 2, the Btu computer performs a calculation based on the difference between the outlet and inlet temperatures to the heat exchanger (T2−T1) and the flow F1 of process fluid through the heat exchanger. This calculation—a measure of the heat transferred to the process fluid—determines the demand of hot or cold fluid needed to maintain process fluid output temperature T1.

Analog computing devices perform the necessary calculations and control can be executed with conventional analog control devices. Additional calculations may be necessary before some variables are combined. For example, the differential pressure signal provided by the commonly used orifice plate is proportional to the square of the flow. A computing element is therefore necessary to extract the square root of the differential pressure signal.

Figure 2 also illustrates feedforward control. The calculation of heat transfer (Btu) rate is "fed forward" to adjust the flow of heating or cooling fluid and change temperature T3. This feedforward calculation anticipates disturbances in both inlet temperature T1 and process flow F1. To provide more stable control of T2, the feedforward signal anticipates the change in heat input required. The magnitude of the feedforward action is usually determined by experimentation and may have to be adjusted periodically, since the heat transfer characteristics of the heat exchanger change with age.

A third control technique illustrated by Figure 2 is cascade control—a technique where one controller adjusts the set point of another controller. The output of temperature controller C1 is fed (cascaded) to the set point of temperature controller C2 through a multiplying device M. Hence changes in process fluid output temperature T2 affect the set point of controller C2 to ultimately maintain output temperature.

The control loops discussed have been applied to continuous processes which operate at near steady conditions with only nominal process or set point disturbances. Therefore, adjustment of the proportional, reset and derivative coefficients is rarely necessary and set point changes are nominal. In a steady, con-

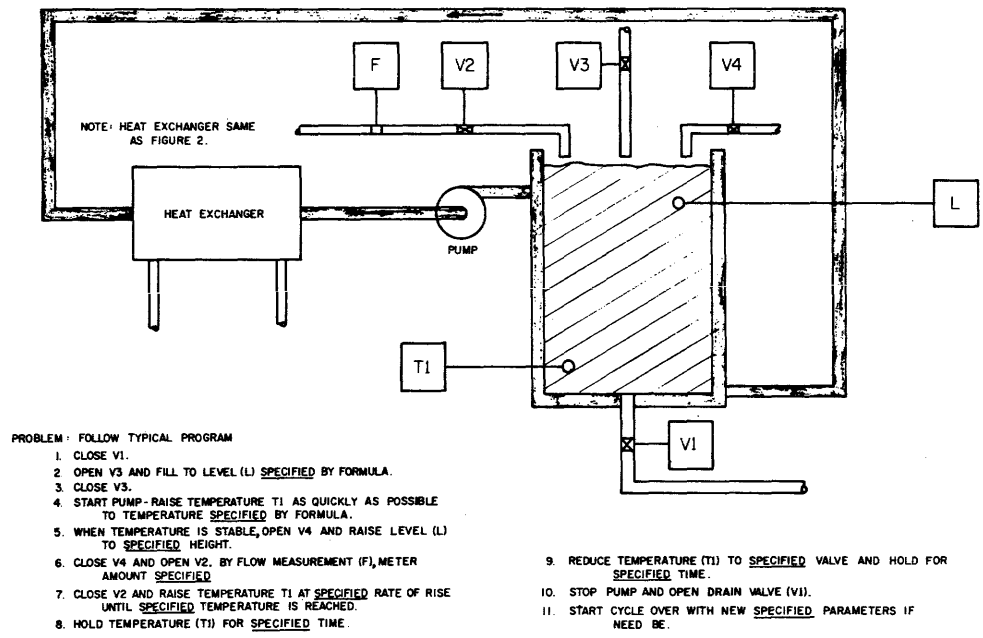


Figure 3 – Simple batch control sequence

tinuous well-behaved process, use of these adjustments would be very limited. Many high production petrochemical processes are in the continuous process category.

Control of semicontinuous processes

Figure 3 presents a process control problem where steady operating conditions are not maintained. This type of process requires a control system which changes operating conditions according to a preplanned event/time schedule. Batch or semicontinuous processes require controlled sequencing because: various equipment must be started and stopped frequently, product requirements change frequently and operating parameters change. It should be noted that most batch or continuous processes still use feedback control, but with programmed changes of control set point.

Figure 3 illustrates a simple chemical reactor. Ingredients are added sequentially and temperature is maintained according to various preset programs to provide the chemical reactions necessary for various products. The reaction within the vessel can vary from endothermic to exothermic during the production cycle. Hence in order to hold a set temperature, the control system may be required to switch from heating the reactor to cooling it when the reaction starts to generate its own heat.

In the typical chemical reactor or mixing vessel, different control sequences may be necessary for each new product. For instance, there may be changes in specified ingredient mix and heating and cooling temperatures and temperature rates of change. Process control problems of this nature require more complex control than the feedback, feedforward and multivariable controls previously described. This control requires programmed sequencing of events, including equipment starting and stopping.

In Figure 3, the control of reaction temperature T1 is basically a feedback control problem. However, the problem is complicated, since T1 must change at the proper times, sometimes in step-wise fashion and other times at a controlled rate. Also, the sequence of events must be readily changed, depending on the intended product.

Combinations of special purpose digital and analog control equipment have been built which satisfy the demands of the discontinuous process. However, the programming of this equipment is relatively inflexible and the control cannot be well-tuned because of the cyclic nature of batch processes. Many of these systems are not used at full operating speed, since the control constants are a compromise.

Applying the digital computer

Digital computers are of significant interest to the industrial process control field due to their ability to store programs, calculate simple and complex control relationships, compute variables which are not directly measurable, monitor the process and take action according to a preplanned schedule. The digital computer easily performs tasks that the analog system finds difficult; it can be easily programmed to adapt the overall control system to changes in process

dynamics, materials, equipment and production demands. Because of this versatility, digital computers are being designed and installed in continuous process plants as well as in batch process plants. Many of the installations use direct digital control techniques on all or some of the control problems.

Table I compares two systems, each using direct digital control exclusively. As shown, a continuous process application in an oil refinery has 530 analog measurements of which 275 are associated with control calculations, the other inputs are for performance monitoring and system operation analysis. Of the 275 control inputs, 180 are used for direct control of simple loops; the remaining 95 are used in advanced control. Therefore, approximately one-third of the 275 inputs associated with control are used to implement multivariable and advanced control techniques.

Table I—Comparison of computer system input/output between continuous and batch process control

	CONTINUOUS PROCESS	BATCH PROCESS
TOTAL ANALOG INPUTS	530	620
ANALOG INPUTS IN CONTROL LOOPS	275	240
CONTROL LOOPS		
SINGLE	180	225
CASCADE	30	70
FEED FORWARD	15	0
DIGITAL INPUT (CONTACTS)	210	1725
DIGITAL OUTPUTS (ON-OFF)	355	1300

Table I also shows the input/output distribution for a large batch control installation currently being implemented by a digital computer system. A comparison of the batch with the continuous process reveals a significant increase in contact sensing elements and on-off control outputs. In order to sequence events, the batch system must sense the status of process equipment and conditions. Also, more devices must be turned on and off. With the batch system, man-machine communication needs also increase. Increased number of push buttons, signal lights and the increased size of digital displays require more digital inputs and outputs.

It is also significant that the number of control outputs (295) can exceed the number of analog inputs (240) in the batch system. This situation occurs in batch processes because the same measurement can be used in control of different control elements and with different control algorithms, depending on the sequence of events and the starting and stopping of equipment.

The philosophy of DDC

With the introduction of the digital computer to the process control field, it became evident that relatively little was known about most processes. Most processes could not be adequately represented by mathematical models which would permit improved process control.

Early attempts at applying the digital computer emphasized supervisory control in which the computer adjusted the set point of an analog controller. In these systems, the analog controller retained the last computer control setting, if the computer failed. On continuous processes, this control was quite satisfactory; in fact, once the system was operating satisfactorily, it made little difference whether the computer was there or not. The operator could still adjust control actions, as he did before the installation of the supervisory computer. This made the process operators happy, but in many instances the process engineers and plant supervisors were not. There was no guarantee that the operators would achieve the optimum control settings for the plant.

What additional advantages did the computer provide? If so desired, the computer could make feedforward, cascade and inferential calculations which would optimize control set points for economic or production considerations. Economic constraints relating to material balance, throughput, inventory, etc., could be developed. In a sense, an economic mathematical model was possible, whereas a process model was still difficult to achieve, due to lack of process knowledge. In addition, the on-line process computer performed other useful work to aid operators, plant supervisors and process engineers: see Table II.

Table II—Some non-critical functions of an on-line process computer

- LOG OPERATING DATA IN ENGINEERING UNITS
- CALCULATE AND DISPLAY OPERATOR GUIDES
- INTEGRATION OF MATERIAL FLOW
- REPORT ON PROCESS STATISTICS - MATERIAL USED FUEL USAGE, THROUGHPUT, ETC.
- CALCULATE AND DISPLAY OR RECORD UNMEASUREABLE VARIABLES SUCH AS BTU RATE, MASS FLOW
- MONITOR AND ALARM PROCESS LIMITS
- RECORD PROCESS EVENTS DURING UNUSUAL DISTURBANCES
- MONITOR AND RECORD CHANGES IN SET POINTS, ALARM LIMITS, ETC. MADE BY THE OPERATOR
- PROVIDE ON DEMAND OPERATOR INFORMATION SUCH AS TREND RECORDING, ALARM STATUS REPORT, LOOP SET POINT AND PARAMETER DATA

Direct digital control was under consideration at the same time that the general purpose digital computer was performing process analysis, monitoring and some set point control.[1] It was reasoned that DDC would reduce the cost of a process control computer by eliminating the cost of the individual feedback controllers. Since the controller merely performs a calculation, why couldn't the computer perform the calculation? Several experimental ventures showed that the DDC concept was physically possible.[2,3] The feedback control law was calculated within a general purpose computer and the resulting signal outputted directly to the control valve.

At first, it appeared that the trade-off between individual loop controllers and a direct digital control (DDC) computer was in the area of 200 loops. There was a hooker, however. This trade-off did not include any provisions in case the computer system failed. For most installations this meant using analog controllers to back up the DDC computer on each loop considered critical.

The DDC equipment was designed so that, if the computer failed, each valve would remain in its last directed position unless backed up by analog control. Critical loops were backed by an analog controller which would maintain loop control on computer failure. Control valves of the other DDC loops were "locked in" at their last output, but the operator could manually position each valve from a console on which he could read valve position and process measurement.

Figure 4 shows two loops from a large system. The measurement M_1, fed to the manual control panel, enables the operator to manually operate valve V_1, in case of computer failure. The loop containing measurement M_n and valve V_n has an analog controller for backup since measurement M_n is fast acting and cannot be controlled manually by the operator.

With the evolutionary history of digital process computer equipment, it is impossible to more than estimate mean time between failures (MTBF). For the smaller digital computers, including input/output equipment, that have been applied to the process control problems, calculated MTBF has ranged from 1000 to 2000 hours. Advances in circuit design indicate that reliability will increase, but reliability statistics on integrated circuits are not yet available. However, regardless of the projections and the calculated claims, the time-shared single computer system will never be perfect and will sometimes fail. Therefore, control security must always be considered on any process installation contemplating a digital computer.

For continuous processes, involving less than 150 loops, it appears that the single computer with set point analog control or DDC with analog backup and some pure DDC on noncritical loops makes the most sense. However, the user must be fully aware that he will give up economic and process control optimization, as well as the functions listed in Table II, if the computer fails. Perhaps most important,

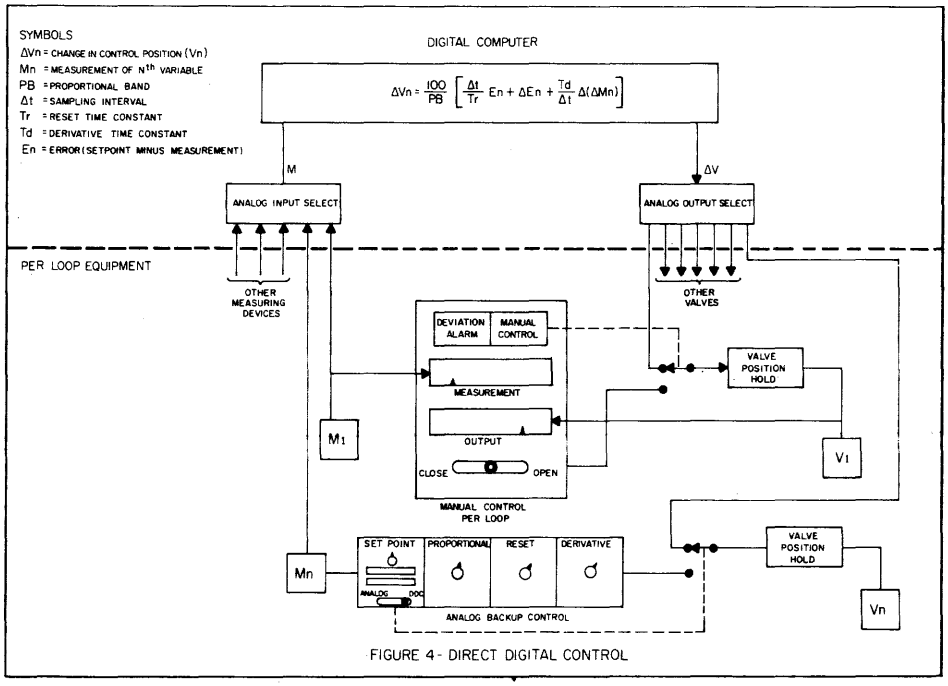


FIGURE 4 - DIRECT DIGITAL CONTROL

Figure 4 – Direct digital control

any advanced control that was dependent upon the computer, such as feedforward, cascade and multivariable, will be lost during computer shutdown.

For the installation where control is not continuous, but where control sequencing is imperative, the use of computer set analog controllers is not sufficient. The computer provides sequencing and logic analysis which must have backup, if process operation is to be assured. The process control problem is not solved by keeping all control settings stationary upon computer system failure. In a chemical reactor, for instance, the contents can solidify or the reaction can "run away," if the process set point is not changed at the proper time.

A parallel DDC computer system

Figure 5 illustrates a parallel DDC computer system which not only provides computer backup but "backs up" the time-shared analog and digital input/output equipment which connects the computer to the various measurement and control elements. It also backs up all interloop controls, as well as all sequence control action.

In addition, this system can continue to perform the noncontrol functions such as those listed in Table II. It therefore permits control to continue even if one computer and/or its time-shared I/O equipment should fail. Note that if any of the time-shared equipment fails, process control is transferred to the backup subsystem.

Table III shows some interesting statistical data[4] which compare the availability of a single computer system with a parallel computer system. The table assumes that the MTBF of a single computer system is the same for each computer subsystem of the parallel computer system. Experience has shown that repair time for various failures, with on-site maintenance personnel, averages between 5 and 8 hours, depending upon the skill of the maintenance personnel, the availability of spare equipment, etc. With the parallel system, it appears that the average repair time can be maintained under 5 hours, since the system incorporates elaborate programs for self-diagnosis to ensure proper transfer to the backup

Table III – Availability – single computer vs. dual computer system

	AVERAGE REPAIR TIME	MEAN TIME BETWEEN FAILURES 1000 AVAIL(%)	1000 OFF[1]	2000 AVAIL(%)	2000 OFF[1]	3000 AVAIL %	3000 OFF[1]
SINGLE COMPUTER SYSTEM	2 HOURS	99.8	17.49 HRS	99.9	8.76 HRS	99.93	5.78 HRS
	5 HOURS	99.5	35.75 HRS	99.75	21.81 HRS	99.83	14.45 HRS
	8 HOURS	95.2	69.55 HRS	99.6	34.95 HRS	99.74	23.04 HRS
DUAL COMPUTER SYSTEM[2]	2 HOURS	99.9998	63 SEC	99.9999	31.5 SEC	99.9999	31.5 SEC
	5 HOURS	99.9987	6.83 MIN	99.99968	1.68 MIN	99.9998	1.05 MIN
	8 HOURS	99.996	14.2 MIN	99.9992	4.2 MIN	99.9996	2.1 MIN

SINGLE COMPUTER FORMULA

$$\text{AVAIL.} = \frac{\mu}{\mu + \lambda}$$

μ = REPAIR RATE (REPAIRS/HR.)
λ = FAILURE RATE (FAILURES/1000 HRS.)

DUAL COMPUTER FORMULA

$$\text{AVAIL.} = \frac{2\mu^2 + \mu\lambda}{2\mu^2 + 2\mu\lambda + \lambda^2}$$

[1] TIME IN A ONE YEAR PERIOD THAT THE SYSTEM DOES NOT PROVIDE COMPUTER FUNCTIONS

[2] ASSUMING REPAIR HAS BEEN STARTED ON FAULTY COMPUTER OF DUAL SYSTEM BEFORE COMPLETE SYSTEM FAILURE

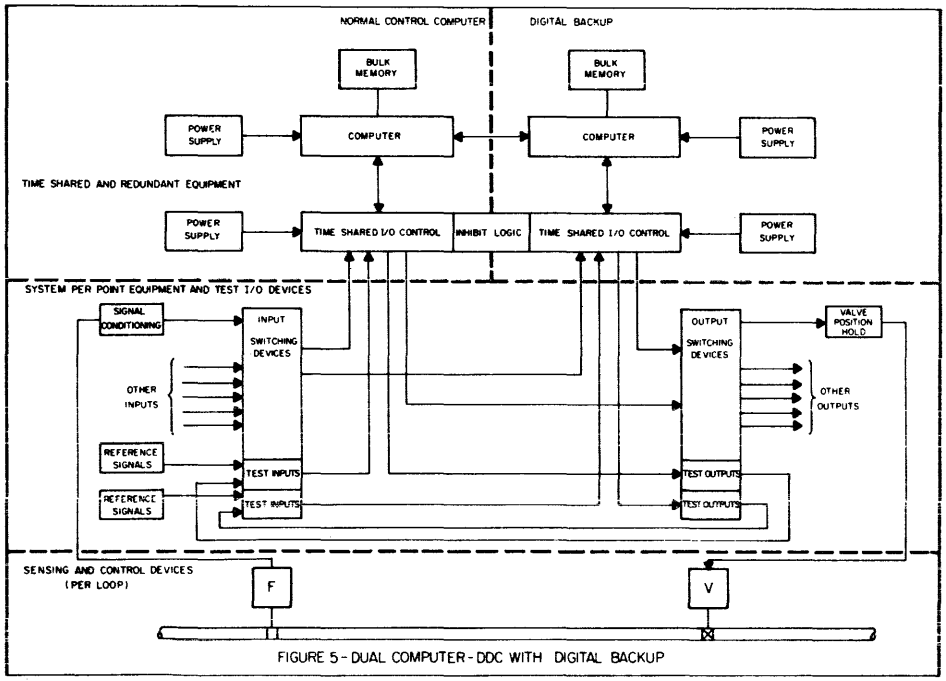


Figure 5 – Dual computer – DDC with digital backup

system. The failed computer subsystem is available for self-checking while the backup subsystem maintains process control.

Systems of this type can be economically attractive since they provide not only the essential control, but the system security essential to batch or start-stop operations. A parallel control processor using direct digital control techniques takes full advantage of the digital computer's process control capability without reservation and compromise. It can include advanced control techniques, such as self-tuning or adaptive control which cannot be obtained with set point control. The parallel computer processing system may provide these features and, in addition, may offer cost advantages over a conventional analog control system for the large continuous process.

For the continuous process in Table I, the computer contains the equivalent of 272 analog controllers. Implementation of a system of this size with DDC and analog backup could exceed the cost of implementation with the parallel or redundant computer scheme.

Input/output equipment

Figures 4 and 5 show that in DDC, as in all control systems, measuring elements and final control devices are still essential. Each measurement is individually conditioned before being fed to the multiplexer of the computer input/output system. Failure of any input or output therefore is similar to failure of a single controller and will not disable other loops. The system should be designed so that failure of any circuit element will not cause the loss of any common power supplies. Also, in case of a power failure, there must be battery backup or a redundant power supply.

Other cautions must be observed in the design of the parallel system interface equipment:

The system must be able to identify and diagnose the fault of any time-shared input/output equipment without disrupting control. The normal control computer and the backup system should both contain several inputs and outputs which can be used for automatic on-line testing of I/O operation, regardless of which subsystem is controlling the process. Some of these test inputs are connected to reference signals, others are connected to output test signals, closing the test loops through each subsystem.

All failed devices must be easily removed for replacement. Any disruption of normal functions during repair should be limited to the few inputs or outputs which share the same printed circuit as the failed element.

There also should be a diagnostic program which verifies correct operation, after the failed component has been replaced.

Output devices, for valve positioning or on-off control, which require power to maintain their status and/or output signal, should have at least a 30-minute battery backup system, in case of system AC power loss.

The system must detect the failure of any time-shared element in the input/output system control logic and automatically switch to digital backup. While operation is in the backup mode, the failed control logic must be electrically isolated and inhibited from operating input and output control devices. Repair can then proceed with no fear of accidental interference with process control.

In normal operation, with the control computer in command, the backup system must continually check its input/output operations to ensure that backup is available.

The inhibit logic must be fail-safe so that its failure will not disturb the system in control. It must be tested automatically to ensure that transfer to backup can take place if a transfer is commanded by a failure detection. If inhibit logic will not transfer the other computer automatically, the system should annunciate that fact and provide an independent manual override which forces transfer of the control of the input/output equipment to the other computer.

Other system design requirements

The system must have a computer-to-computer communication link which continually updates the backup program data and status on a periodic fixed time basis. The backup computer thus receives dynamic operating conditions within a short time period (in the order of seconds for a batch process).

Any program changes made on-line while the control computer is operating the process must be transferred to the backup control computer, at the same time. This updating must include operator changes to control settings as well as any on-line program changes.

A bulk memory must be used on both computer systems to retain the many formulas and programs that may be required. Bulk memory can also contain interpretive programs to simplify construction of a batch program, diagnostic programs for fault detection and programs to aid maintenance. Sophisticated man-machine communication programs, which involve lengthy message storage, can also be included.

Diagnostic programs for the computer-to-computer communications link should test for link failure, annunciate the failure and command the changeover to the backup system. A program system permits updating and on-line diagnostics while time-sharing the real-time programs in bulk memory.

There should be a system procedure and a system diagnostic program to assist in rapid repair of a failed subsystem. Another procedure and program is required to transfer all operating programs from the backup subsystem back to the repaired computer, without interfering with process control.

When the backup system is not on control, it is available for program compiling, debugging and problem simulation using the test inputs and outputs. It must also perform diagnostics to ensure operation is correct for takeover if necessary. When backup computer takes over process control, these programs are discontinued.

CONCLUSION

By using DDC with complete input/output control and computer backup, the parallel computer processing system permits unrestricted application of computer control techniques. It takes full advantage of the logic and computational ability of the digital computer, whereas a computer system which depends on analog set point control or analog backup cannot.

The parallel control computer system program storage ability, together with backup of logic control, program sequence and formulation, makes it ideally suited for complex batch or start-up and shutdown applications.

Complex continuous control systems would also benefit with this control system. Built with state-of-the-art electronics, the system should challenge the economics of computer set point control and single computer direct digital control with analog backup.

REFERENCES

1 J W BERNARD J F CASHEN
Direct digital control
Instruments and Control Systems Sept 1965

2 E VANDER SHRAFF W I STRAUSS
Direct digital control - an emerging technology
Oil and Gas Journal November 16 1964

3 J W BERNARD J S WUJKOWSKI
DDC experience in a chemical process
ISA Journal December 1965

4 R H MYERS K L WONG H M GORDY
Reliability engineering for electronic systems
John Wiley and Sons New York 1964

Real-time monitoring of laboratory instruments

by DR. PAUL A. C. COOK
Celanese Research Company
Summit, N. J.

INTRODUCTION

The earliest investigation into the feasibility of automating data collection at Celanese Research occurred in 1961 when conversion of Instron tensile testing data to digital form (punched paper type) for later processing by computer was considered. The first workable systems for aiding in reduction of Instron tensile testing data were developed by DuPont, initially recording data on magnetic tape for later play-back into a computer.[1] Subsequent systems used shaft encoders on the Instron cross-head drive and recording chart pen to digitize data for punching on paper tape.[2] The more modern systems by-pass the slow and cumbersome paper tape and read data directly into a small digital computer. Instron now markets a digitizer for producing punched paper tape. Control Data Corporation, who designed the first computer based systems, and several other manufacturers of small computers supply systems to both digitize and analyze Instron data.

Other types of laboratory instrumentation notably gas chromatographs, mass spectrometers and others have also been equipped with digitizers and computers.[3,4,5] Computer technology has now advanced to the point where it is often no more expensive to interface directly to a computer than to produce an intermediate output such as paper tape, cards or magnetic tape. It is also true that even small computers today operate at such high speeds that many laboratory instruments can be monitored in real-time.

Hence, the system at Celanese Research is designed to be expandable to 50 instruments or more.

Incentives

A major incentive existed at Celanese Research for saving on manpower and improving on recovery of data in the physical testing area. Under peak conditions there are four Instron tensile testing machines operated on a two-shift basis turning out as many as 1000 samples per day. A typical stress-strain curve, shown in Figure 1, illustrates the complexity of data obtained. As many as twelve useful and meaningful parameters can be derived from the curve. Prior to installation of the monitoring system, there was simply not enough manpower available to completely analyze more than a small percentage of the stress-strain data generated.

For the manual system only terminal breaking properties (tenacity and elongation) are reported for *all* tests. Modulus (the initial slope of the curve) and yield point, which are better indications of the true tensile properties of a material were determined by hand calculation for some samples. Work to break, i.e., the area under the stress-strain curve was also obtained manually or with mechanical integrators for some samples. The rest of the potential information was for the most part ignored although for a small percentage of important samples other parameters were manually determined and an average stress-strain curve was plotted by hand by overlaying a series of Instron charts and tracing by eye an average curve.

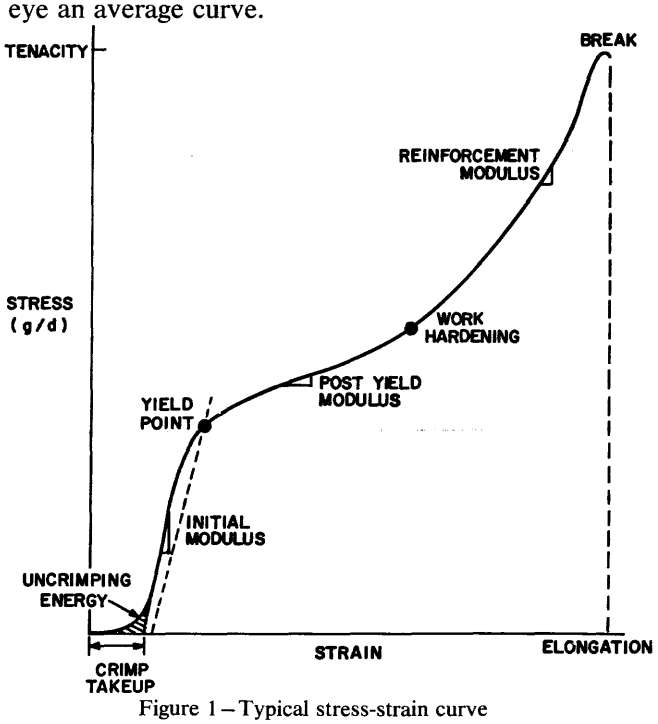


Figure 1 – Typical stress-strain curve

From this description, it is obvious that there were two main incentives for a monitoring system, first, savings in man-power formerly required for data reduction and second, the recovery of roughly 80% of available information from the tests which was being lost because of lack of manpower.

General system description

A block diagram of the system is shown in Figure 2. Starting at the instruments, signal conditioners which provide any necessary amplification and filtering connect the analytical output to the multiplexer. The higher speed multiplexer switches the various instruments to the analog to digital (A-D) converter. The A-D converter output goes through a control unit to the memory of the computer. The control unit also handles inputs from operator control panels which are located at the instruments. Standard hardware for the system consists of a main computer (CDC 1700 with 1.1 μ sec cycle time, 8K of core storage, 16 levels of priority interrupt, parity and memory protection), card reader, paper tape reader and punch, line printer, disc mass storage, teletype, and digital plotter.

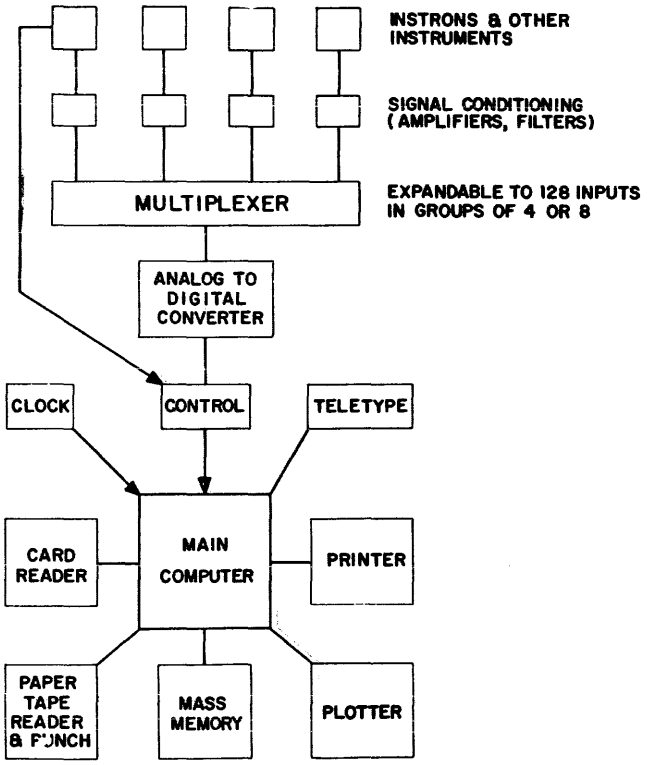


Figure 2 – Block diagram of instrument monitoring system

Instron application

Interfacing of a stress-strain curve requires reading the stress measured by a strain gage plus some measure of the rate of strain. While many Instron systems have been built using shaft encoders on the pen recorder to convert the stress output to digital form, these systems suffer from two main disadvantages:

(a) Their speed of response is limited by the pen recorder.
(b) There is invariably some error introduced by the pen drive system.

Hence, the Celanese system was designed to take the pre-amplified strain gage signal thru amplifiers and filters to multiplexer and A-D converter. The strain rate is taken from an internal clock timer since the Instron drive system has ample regulation to ensure a constant strain rate.

Specially designed operator consoles (Figure 3) provide for 44 digits of information to be entered by the operator plus ten pushbuttons and three indicator lights for control of tests. The consoles despite their higher cost were chosen instead of teletype keyboards in order to reduce the chance of operator error. The fact that all printout is handled by a single line printer also influenced the decision to go to consoles. It should be noted that development of a low-cost, modular operator console would make this approach more attractive for many instrument monitoring systems.

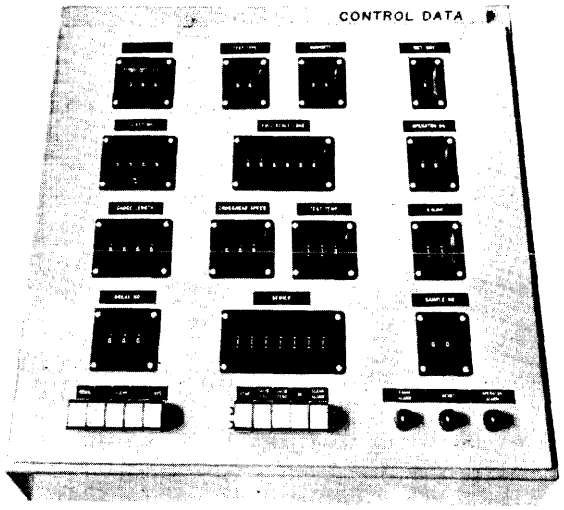


Figure 3 – Tensile test operator console

System operation

Before mounting a new sample the instrument operator inspects the "Ready" and "Active" indicator lights to verify that the system is on-line and the results of the last test have been transferred to the output buffer. Then fixed parameters are set into the thumbwheel switches in the console. The

following items may be input through the console:

(1) Operator number
(2) Sample number
(3) Work ticket number
(4) Break number
(5) Full scale load
(6) Crosshead speed
(7) %Elongation Scale length for plotting
(8) Denier
(9) Gauge length
(10) Test Type
(11) Test Temperature
(12) Humidity
(13) Wet or dry
(14) Spare

The recorder and input hardware may be calibrated if desired prior to mounting any sample.

After mounting sample the operator presses a start button which initiates automatic operation of the instrument. The interrupt from the start button causes the thumbwheel switches to be read, actuates a relay starting the crosshead motion and sets the proper bit in the storage word indicating which instruments are in operation. Stress data are sampled via the multiplexer and A-D converter every 4 milliseconds for all instruments in operation. The monitor subroutine has a built-in digital filter to eliminate high frequency noise from the stress data. Data are accumulated into scratchpad areas in core during the actual testing.

Termination of the test is detected by the monitor program when the stress drops rapidly after the specimen breaks. At the end of any test the operator has the option of rejecting the data taken if there has been an obvious malfunction of the testing equipment. If the test has run satisfactorily, pressing the "OK" button initiates calculations on the current data. If the operator discovers that incorrect parameters have been set in the thumbwheel switches, he can reset proper values and request a reading of the thumbwheels plus a recalculation by pressing another pushbutton.

The operator signals the end of a series of breaks by still another pushbutton which initiates statistical analysis of the data, transfer of data to the output queue for the line printer, punching of statistical data on paper tape and plotting of an average stress-strain curve plus the envelope of maximum and minimum curves on the digital plotter.

Alarm lights on the control console indicate when the operator has taken some illegal action or when there has been a hardware failure. Error messages are printed on the teletype unit at the computer.

Software

As is evident from the description of system operation, software design presented several unique problems. Since all output except for error messages was via a single line printer and results for a series of tests for a given instrument were to be grouped together, it was necessary to set up scratch-pad areas on disc for each instrument and an output queue on disc for handling printouts. In addition plotting an average stress-strain curve for a series of tests required generation and storage of a set of plotting points. To prevent interference between instruments, it was also necessary to provide an output queue for the paper tape punch. In order to accommodate all of the data storage areas required, it was necessary to use essentially all of the 8K of core memory and about 30% of the 1.5 million words of disc storage.

The program is modular to facilitate future expansion. The monitor which handles the sampling of the instruments every 4 milliseconds was designed to operate as efficiently as possible and actually uses only about 100 microseconds. Processing of input data and handling of input-output is much less frequent and uses less than 2½% of central processor time. This means that less than 5% of central processor time is used for monitoring the initial four instruments and leaves room for many more instruments to be added.

The data analysis program has several unique features the most important of which is the method by which different stress-strain curves are handled. In contrast to many previous systems, a single program handles all single fiber tests. This is remarkable when one considers the wide variety of shapes of stress-strain curves encountered in fiber testing in a research laboratory. The major problem in designing a comprehensive analysis program was defining a unique algorithm for finding the initial modulus. The procedure is as follows:

(a) the program searches for a straight line portion of a given minimum length and noise level,
(b) if a straight portion can't be found, a point of inflection is searched for, and
(c) failing either (a) or (b) the initial slope of the stress-strain curve is taken as initial modulus.

In cases (a) or (b) a correction is made for any initial crimp or preset in the fiber. The algorithm is not perfect but works in better than 80% of the cases tested. Because of the back up provided by the digital plot, it is easy to spot cases where the algorithm has failed and modulus values are meaningless.

The initial program has four options to handle

(a) fiber testing,
(b) fabric or other heavy material testing,

(c) yarn testing and
(d) crimp testing.

The next phase of the project will add facilities for cyclic testing of elastic materials with on-line control of the testing machine and cyclic testing of materials at different temperatures for determination of glass transition.

Future expansion

A special interface for a high resolution mass spectrometer is currently under study as the next step in expansion of the system. Interfacing the mass spectrometer requires monitoring of three instrument parameters: ion current, ion accelerating voltage and magnet current. Automatic ranging is provided for the ion current which may range from several millivolts up to several volts. Accuracy is 1 millivolt on the lowest scale. Accelerating voltage and magnet current are measured to an accuracy of 1 part in 2000. Operator consoles similar to those used with the tensile testers will be used.

The software is designed around the same instrument monitor with 4 millisecond intervals between samples. Peak detection for the mass spectrum is under program control. As peaks are detected, peak height (mass to charge ratio) and peak position (mass number) are stored on disc. At the conclusion of a test all peak heights are normalized to 100% for the largest peak. Normalized spectral data are punched on paper tape and printed on the line printer. For analysis of unknown samples the paper tapes are read back into the computer for processing by a program that operates in the "background" to the monitor program. Alternately the analysis program may be run in batch mode when the instruments are not being monitored or on a separate large scale computer.

When the mass spectrometer is added to the system, a system will also be provided to permit compiling and running FORTRAN programs while monitoring instruments. This will be a "real-time" FORTRAN so that new instrument programs as well as data reduction programs can be compiled. Also under study are proposals for tying in instruments at more remote locations using phone lines for transmission of analog data and returning results to a teletype unit. Other types of analytical instruments that are well suited to on-line monitoring are:

(1) Gas chromatographs
(2) Spectrometers
(3) X-ray diffractometers

Monitoring and data logging for several pilot plant operations will also be added to the system.

REFERENCES

1 G O PATTERSON JR T D MECCA
Journal of Applied Polymer Science Vol V
pp 527-532 1961

2 J D GRANDINE
Paper presented at symposium on
Analytical methods in the study of stress-strain behavior
Boston Mass Oct 29 1960

3 I LICHTENSTEIN
Paper presented to Instrument Society of America
meeting Oct 24 1966

4 ANON
Chemical and Engineering News p 48 Nov 1 1965

5 ANON
Chemical Week p 52 April 30 1966

Humanizing industrial control software

by J. B. NEBLETT and D. J. BREVIK
Honeywell Computer Control Division
Framingham, Massachusetts

INTRODUCTION

Process control by computer is common enough today to justify postulating an industrywide basic software system to aid in the implementation of control systems. There have been many successful industrial control systems; however, the main emphasis of this paper is directed toward the problems that prevented some other systems from attaining really successful operation. The reasons presented are mainly concerned with the software of programming aspects and not with the suitability of hardware or the basic financial justification for the computerized system.

Let's assume that the basic application justifies the installation of a computerized set of hardware, that the hardware itself is matched to the process problem in general and that the basic hardware is operational and can be maintained operational. What then is the underlying problem in getting a system of hardware-software and application knowledge meshed together into an operating process control computer system?

The "meshing" process is, of course, the programming of the system, and it has been common to regard this task as a mechanical one. Functional specs are laid down, then implemented. The resulting program(s) is checked out and put into operation.

The whole system now goes into service and usually, although it meets specifications, falls short of the user's expectations as an automatic controller (or operator) of his plant. Furthermore, he usually finds that the system is surprisingly difficult and expensive to change to meet the requirements spelled out by using it for a while.

This whole experience is analogous to taking a new graduate in engineering, assigning him a major design responsiblility and finding his performance falls short of expectations. The point, of course, is that the new graduate needs a period of TRAINING between leaving college and the assumption of heavy responsibility.

The authors believe the approach to programming a control computer application should recognize this TRAINING phase as being just as desirable for the automatic operator as for the human operator. In the following discussion the reasons for this view are explained.

The supplier's software

The programming language in the past has been very primitive. The basic software approach was:

1. To train the user to speak the language of the machine whether it be basic octal or symbolic assembly language.
2. To cause the user to learn the idiosyncrasies of the particular piece of equipment that he was attempting to apply.
3. To be sure the user had an intimate knowledge of the process to be controlled.

Systems in the past have reflected a notorious lack of a systematic approach to the programming problem. Software systems in the past reflected a definite lack of consideration for the inevitable reprogramming requirement common in process control applications. The poor documentation associated with software systems intensified the problem of the programming and reprogramming as well.

We believe that these problems can be overcome if both the supplier and the user regard the hardware-software system as resembling the characteristics of the human operator to be trained to do the job. This constitutes a large measure of common ground in that both the supplier and the user are familiar with the training of human beings and the terminology and techniques used in such training. The supplier wants to amortize the cost of development of his software over a maximum number of systems while the user wants to apply software

without major modification to that basic software. Few suppliers have the experience of their customers in the customers' applications. They cannot foresee every use, every possibility, without risking either extensive specialization or overgeneralization of software. Few users agree entirely with the suppliers' conclusions that are implicit in the software design. A common meeting ground appears to be in regarding the computer as a human operator.

Software designers of the past have hinted about "humanizing" software. Rarely, if ever, have they admitted a whole-hearted swing toward identifying human operator characteristics with the computer and its software. Our contention is that software should be humanized to the fullest extent possible and that this tactic would be the most common meeting ground of both the user and the supplier. We believe that a remarkable degree of humanization is desirable and possible.

Many times in the past, problem-oriented languages have been designed directly toward one specific type of user, e.g., the civil engineer, the simulation engineer, etc. The field of process control is so diversified that a supplier, without risking overspecialization, cannot afford to concentrate on only one type of process control or to generate problem-oriented languages dedicated and restricted to that type. There must be a common meeting ground where the supplier can make the equipment usable for the great majority and the user can economically make that equipment fulfill his special-purpose requirements.

This approach must include a basis for generating the problem-oriented language in much the same way that a supervisor would train his operator to think, speak, and respond in terms of the problem at hand.

Composite software for the composite user

Who is the User?

The supplier would design the software as a composite of the best operator features known. He would also design it to be used by the best composite user known. This does not assume that the ultimate user is an absolute genius in all aspects of solving the problem but instead assumes that the user is actually many people—the process engineer, the production manager, the plant dispatcher, and perhaps even the accountant. To put it simply, the software is usable by specialists typical in the process control field. The user takes an engineering approach to the problem at hand and recognizes the potential of improvement within the process. While he is not necessarily an expert in computers and instrumentation, he regards the computer and programming as a means to improve his operating process economically.

The user has a general understanding of programming or, at least, is capable of absorbing the principles of programming. He has had experience in personnel training and evaluation. He works with people and is capable of working with new people introduced to his environment. He is not averse to working with the computer if necessary and has no mental blocks against learning how to deal with the computer. He rarely has the time to learn all the intimate idiosyncrasies of a particular computer much as he rarely has the opportunity to learn all the intimate idiosyncrasies of his process. He depends upon others within his sphere to provide him with straightforward answers. He does not want to be an expert in their field. He is likely to regard the computer as an interloper into his particular domain unless the computer is easy to converse with, produces meaningful results, and does not usurp his job.

What is the user's problem?

The user's problem is to cause a digital computer to sample, interpret, and act upon information obtained from the process or from operating or management personnel concerned with the process. Presumably the application of the computer results in an improved economic picture.

His problem should not be to master a new discipline. He should not be required to learn the esoteric information concerned with programming, computer maintenance, and the peculiarities of the particular computer. We suspect he's in enough trouble keeping up with the changing demands of his own discipline.

Instinctive approaches to solutions of problems

An instinctive approach to the solution of the user's problems is based on making possible his communication with the computer as if it were a human person or at least something with many recognizable human characteristics.

While it is clearly not possible in the present state of technology to program the digital computer to act like a human being who is completely conversant in the specialty of the user, it is possible to program it as an "ideal" operator that can:

1. Understand currently known engineering facts of the process,
2. Understand how to apply good operating practices,
3. Make sound engineering decisions, and
4. Make management decisions regarding the economical approach to production.

Traits are what you look for in hiring an operator. These are the characteristics of the individual that indicate how well he can learn new abilities. On the other hand, most abilities are bonuses when you are interviewing an operator. An operator or a potential operator who has the desirable traits and, in addition, has some desirable abilities is a good prospective employee. Similarly an operator with the desirable traits but without the abilities is still a desirable prospective employee. Conversely, a candidate with abilities but without the correct traits is a poor prospective employee. The worst case is an operator who has bad traits and bad abilities (that is, preconceived notions).

Similarly, the software system accompanying an industrial control computer must be judged on its traits rather than the abilities taught to it by the supplier. The desirable traits are not always self-evident. Among them should be included the ease of trainability so that the ideal operator can learn the abilities required for his job. Other traits, such as honesty, integrity, reliability and so forth, are essential to the operator but are frequently overlooked or taken for granted in industrial control software.

With regard to these traits, consider the points of evaluation of a human operator during the pre-employment interview, training, day-by-day operation, and finally, the qualifications for advancement. Important qualifications of an applicant for an operating position would be:

1. Adaptability to the working environment.
2. Basic knowledge of the particular process.
3. Capacity for learning new procedures.
4. Logical rather than emotional judgement in response to process control situations.

The candidate must be reliable and consistent and must exercise discretion when called for. He must have a high degree of retention, and must perform his work accurately. It goes without saying that he must possess a sense of time, and along with this he must have a high sense of urgency in categorizing the relative importance of his functions. In other words, he must recognize that he can't do all things simultaneously but must schedule his activities when many things have to be done almost simultaneously.

While the operator is being trained, he must have a certain rapport with the instructor so that he can indicate lack of understanding when necessary. He should be honest and indicate that he is reaching the limit of his capabilities and is approaching saturation. Along with this, he should be easily retrainable so that when the instructor tells him to forget a procedure which may be erroneous, he will forget and will be able to learn an alternate procedure. He must be capable of learning at a reasonable rate of instruction so that the instructor does not get frustrated and impatient with a slow pupil.

There may be many teachers, many instructors training this operator, so he must be able to recognize gross conflicts of information and facts fed to him. Furthermore, there may be a hierarchy of teachers. After all, we don't want him listening to the janitor in preference to the production supervisor. He must exercise discretion about the information he is given; certain facts may be proprietary, not for everybody to know. Therefore, he must have a sense of secrecy. In short, he must have a keen sense of the meaning of an organization chart of authority.

If an operator becomes ill, he should notify his superior and perhaps provide some degree of self-diagnosis. This, of course, allows the supervisor to take alternative action, perhaps even replacing him temporarily. An absolutely ideal operator would be able to notify his co-workers that he was going berserk.

If an unnatural operating situation has arisen, one that was unforeseen, he should be able to tell his supervisor what actions he had taken up to and during the emergency situation. This enables an analysis or a post-mortem of the event in order to improve future operating procedures.

He must be able to accept responsibility as it is handed to him. While it is not reasonable to expect him to assume complete responsibility at the very beginning, it should be possible to gradually increase his responsibility as his skills evolve. As a corollary, if he has not been able to absorb a new responsibility—perhaps because of poor training—he should be perfectly willing and able to surrender that responsibility and to be retrained in that function.

Training and developing the operator

Given an operator with these traits, the user must train and develop his abilities methodically. There is a step-by-step procedure by which the instructor sets the pace according to the reactions of the student while learning, with a continual, informal evaluation of his performance on the job at each step of the way.

How might this be accomplished?

First, the new operator might be shown the control room and the basic nomenclature and jargon used in the plant explained to him. Certain basic procedures are described—such as how to read the instrument panels, how to contact the laboratory for analysis results pertaining to his job, whom to contact for production goals. At each of these steps, he is tested to determine whether he has learned properly—which may be a reflection upon either the operator or the teacher.

After being told how to read the instruments, he is tested and asked to read a few instruments while the instructor notes his performance. Likewise, as each step of data acquisition is explained to him, he is tested informally to determine whether he is truly following the correct procedure. The first step of this teaching process is to make him aware of his surroundings, the sources of basic information available to him and their uses. Perhaps he has not yet been told what to do with this information but only how to get it.

After this, he may be asked to record periodically or log the information he receives. It may be necessary at this point to explain to him the meaning of the word "log." This is possibly the first piece of responsibility he must assume. After he has done this for a period of time, his performance is evaluated: Is he performing the procedure that was explained to him? Is he extemporizing? Is he doing precisely what was expected of him? If not, why? Eventually he masters that skill, perhaps after a short period of retraining if some deficiencies have been found.

Having mastered one skill, the operator starts to learn the next one and his responsibilities will be increased. The next step may be one of accessing the information he has been taught to get, and comparing it with some rudimentary limits or constraints of operation and reporting violations to his immediate supervisor. Note that he is exercising nothing more than mechanical judgment rather than control or engineering judgment.

The skill he was taught in warning his supervisor when constraints were violated depended upon his previous mastery of an old skill, that of accessing the data needed. Throughout the learning process, new skills are usually dependent upon mastery of old skills. At every point, when a new skill—or procedure—is being taught to this operator, he is tested to make sure he is absorbing the instructions properly and is performing according to the standards set for him.

At any time, new data may be introduced to the operator such as a newly installed recorder, but the skill associated with the data remains unchanged. At any time, the operator should be capable of understanding that new readings or constraints or data are being introduced to replace those he once knew. As a general rule, the procedures he uses, once learned are quite inflexible. This does not preclude the possibility that an operator must be retrained to new procedures.

As the ideal operator progresses, and learns the basic skills of his trade, it is desirable to go beyond the mere mechanical manipulation of the process into a more detailed explanation of the phenomena taking place. We want him to become a practical engineer. This training would be accomplished by a senior operator who has had years of experience in knowing the idiosyncrasies of the process, and who can give a good layman's explanation to the operator. This senior operator knows the control characteristics and can anticipate the reaction to any change he imposes upon the system. If the trainee were truly the ideal operator with an engineering background, an engineer could train him in engineering terms.

Throughout this training process a methodology is developed. The operator is told procedure, is tested and if found satisfactory is given more responsibility which requires exercising the newest skill as well as other skills previously learned. This cycle is repeated until the operator is fully capable of performing what was originally expected of him. And also, at any point in his training or even after he is fully trained, he must be retrainable with a minimum of difficulty.

Traits inherent to the digital computer

There are certain traits desirable in the operator which are inherent to the digital computer. These are:

1. Reliability.—Computer technology has advanced to a state where extremely high reliability should be expected in any solid-state computer and peripheral real-time equipment.
2. Accuracy.—The inherent accuracy of a digital computer is generally superior to the sources of information from instrumentation and the typical accuracy attainable by normal operating personnel.
3. Retention of Information.—The computer is superior in its capability of retaining details associated with information. Typically, however, a human being has more overall capacities for information retention.
4. Speed.—The computer will typically be able to react and solve a problem more rapidly than a human operator.
5. Diligence.—A computer system is inherently more diligent than an operator and is unlikely to be found sneaking off to a smoking area at a critical time in the operating cycle.
6. Consistency.—A computer reflects no quirks or aberrations, but consistently executes what it has been told to accomplish. A human operator may be inclined to experiment without authorization; a computer system has no such ego and will consistently do what it has been told to do.
7. Flexibility.—A computer system which operates with a stored program has a flexibility somewhat

similar to that of the human memory and mind.

8. Emotional Stability.—The computer has no family problems to interfere with its control of a process system. The computer bears no grudge against the boss or ambitious fellow-operators.
9. Raw Senses.—Synonymous with the five senses of a human operator, the real-time computer has a capability of sensing external stimuli such as time, interrupts, raw instrument readings, and indicators from the process.

There are certain desirable and valuable traits of a human operator which are very weak or lacking in a computer system. They do not preclude the utilization of a computer as a substitute for the operator.

1. Curiosity.—A good operator is typically curious and probing to find out more about what is going on in a surrounding situation. A computer system is curious only when it is told to be curious and about those things that it has been instructed to probe further. Curiosity is valuable in an operator because questions associated with the operation of a process may be the first reflection of a possible abnormality within the process operation.
2. Heuristics.—The human being has the latent capability of learning which a computer system does not have. Although there are currently very active investigations into heuristic type software, it has not yet reached the state where we feel it is practical to include it in the discussion of a computerized process control. The heuristic capability of a human operator provides a capability of self-training which is not inherent in a computer system. A computer learns only what it is specifically taught and told.
3. Limited Communications.—Over the years, human beings learned to communicate not only by words or tone of voice but by the facial expressions and gestures that are typical in a conversation. The computer system is somewhat stilted in its communication and consequently cannot fully match the dialogue attainable by two human beings.
4. Self-Diagnosis.—A human being can usually detect when he is getting sick whereas a computer often reacts in an instantaneous and catastrophic fashion to a malfunction similar to the effects of a heart attach on a human being. Because of the nature of the malfunctions associated with a computer, it is often meaningless to advise a supervisor of pending "immediate" diaster, but is more logical to provide redundancy and/or external holding circuitry which can buffer instantaneous malfunction from the process. This allows a more leisurely take-over of control by a human operator.
5. Mobility.—In case of disaster, such as a fire within the process, an operator can run out and escape; but a computer is immovable and possibly will be destroyed in the blaze.

The above discussion is concerned with the basic inherent traits of a computer, both strong and weak. Given these traits and a basic command repertoire, it is possible to program certain *abilities* into a computer hardware-software system. These abilities should be analogous to the abilities we have described for a human operator.

We are about to suggest a computer hardware-software system with traits and abilities of an ideal operator. To distinguish between a human operator and the thing that is the computer software system we have chosen the term "android" for the latter. Strictly speaking, what we are describing is closer to a robot, but the term "robot" has certain overtones of sensationalism in the yellow journals. On the other hand, the term android is somewhat of a misnomer since the dictionary definition is that an android is an "automaton of human form." An automaton is a terribly clumsy form of human activity; a robot is a more sophisticated form, while in the science fiction literature magazines the android is the highest form. We have no pretentions of describing the highest form of android, but wish to avoid the visualization of a mechanical monster from Mars clanging down the aisles. We wish to avoid the typical connotation of a robot with mobility that is able to walk from place to place and having the manual dexterity typical of human beings. Our term, android, covers only certain mental characteristics or traits common to a human operator.

The abilities of the computer-operator

Traits alone are not enough. Traits are a measure of the character of the individual as well as the character of the computer. Abilities, on the other hand, are training patterns that take advantage of the traits. As an example, a typical trait of the human being is either honesty or dishonesty. An honest man can be taught how to operate the cash register and become a productive employee. Teaching the same ability to a dishonest man is a step toward financial disaster. Previously, we have enumerated the basic traits which are looked for during the interview of a candidate for an operator's position.

If we puruse the idea of identifying human characteristics with the digital computer in order to more

effectively simulate the human operator, then we must consider the abilities that are prerequisites for the basic operator.

Communicating with the computer system

The language of communication to the computer system should be that of a high-level compiler. A compiler is specified because it is possible to slant language to the user rather than to force the user to learn the particular language of the machine. While machine language has its place, we do not consider this the natural language used in industrial process control by an engineer or a chief operator talking to a human operator. As with any compiler, the user must have the ability to declare a form of data. Data takes the form of real, integer, Boolean, and logical, plus additional forms of analog, digital, BCD and the unique forms found in process operations. The data declaration capabilities in a compiler provide the *nouns* which can then be utilized to train the operator in the process. Future reference to previously defined data may then be made in a nomenclature familiar to all associated with the process.

Other nomenclature peculiar to a process where verbs can best identify the procedures of the process, the supplier should apply a basic set of verbs common to all users; however, the capability of defining additional verbs must be a latent part of the compiler system. Procedures consist of nouns and verbs and may be defined by another verb. Once defined as a procedure, future reference to the defined procedure can be made in terms of the new verb.

As the new operator learns the names and procedures associated with a process, so the computer system has this ability to learn such details required to operate the system. This simplifies future communication and once learned, permits much simpler reference to these previously taught nouns and verbs. Certain verbs are basic to the system, such as DO, PRINT, CONTINUE, IF, and others found in a language such as FORTRAN. New verbs imply a sense of urgency, the ability to wait or pause and proceed, and the ability to suspend at any point until the completion of a response to an action. In addition, the language must provide the ability to evolve toward a problem-oriented system. We do not claim that the language modifies *itself* toward a problem-oriented language but that it enables the user to think in a problem-oriented fashion.

There must be a means of indicating the relationship of urgency between all procedures. The computer—much as the operator—cannot be expected to sort out the relative emergencies within the system, but must be given indicators by the teacher during the learning process. These should not be rigid parameters of urgency, but relative instead. As new procedures are taught, the urgencies of all procedures must be sorted out by the computer system rather than have the user go through all the old procedures and redefine the urgency relationship of all.

It is natural to assume the operator has a strong sense of time and is capable of looking at a clock. Similarly, the language should implicitly assume a knowledge of time. Statements such as, "at 1100 hours do the following," etc., should be allowable. Statements similar to "every hour on the hour" should also be allowable. And statements such as "delay ten seconds" should be permissible. The sense of time implicit in the system should be the same sense of time a human being feels. The user should not have to supply long and tedious calculations to represent a next call time for the procedure.

A human operator is quite capable of understanding the context of a sentence such as "do this and, in the meantime while it is completing, go off and do something of less urgency but keep your eye on the completion of that event that you started because when it's dome go back to the more urgent activity." It is a natural way of expressing what to do when you reach an impasse. An analogy might be "start the pump motor and when the tank is full turn off the pump; but meanwhile take care of your other duties." Here the operator is assumed to have a sense of completion of an event. He does not have to sit there and stare at the tank to determine when it is full. He may have to glance periodically at the level indicator and, when it begins to approach fullness, turn complete attention to the tank; but while his full attention is not needed as the tank is beginning to fill up, he can temporarily divert himself to other duties.

Implied abilities

The computer system should be self-documenting. An operator can be called upon to repeat what he has been told. The computer should provide the same ability. A computer should be able at any time in the future to retell what it has been told. Obviously, it should retell in the language of the user and not in the natural language of the computer, that is, machine language. Everything that has been told the computer to date should be readily accessible in the user's language as well as in any internal form the computer may choose to use. However, a first offering might be hard copy produced at source time and consisting of a listing of the source program along with the date and the name of the user. This places some burden on the user to classify and collate the papers.

The ultimate offering, though, is complete self-containment in the computer system.

A human operator is self-organizing, he does not have to be told in which part of his brain to store information. Such an instruction can be as meaningless with computers as with human operators these days. A self-organizing computer system is required if the computer is to give any indication of potential saturation. Otherwise, the users are left on their own to update memory maps, and make intuitive judgments as to the degree of saturation of the system.

The ideal operator can be trained on the job and does not have to be pulled off of productive work to go to a classroom. It is the same with the computer. What is known now as background/foreground processing is a requirement.

Background is the place where new skills or procedures are taught to the computer system, while foreground is where previously learned skills and previously defined responsibilities are executed. The operator is trained methodically and is given increasing degrees of responsibility as confidence in him in gained. A new procedure introduced to a computer is debugged methodically. A self-documenting system which states unequivocally the exact procedure explained to it must also be able to state the sequence of execution that actually occurs. This documentation must also be in the user's language.

When an operator is given an explanation of a new procedure, he is generally stepped through it by the teacher to see if he really understands. He may be asked to repeat each step as he does it verbally so that the teacher may see whether the operator really understands what he is supposed to do. The first few times an operator attempts a procedure he is not allowed to actually influence the process. He goes through the motions and simulates what he is supposed to do. This simulation is expressed by some verbal comment such as "and now I twist this valve." This is simulation of the operation, and this software system should be capable of such simulation.

The teacher may wish to present a test case or test inputs to the operator to determine if he is doing the proper thing. He may give a typical problem, such as "suppose the temperature of the vessel reaches 500° what do you do?" The operator then is expected to go through an explanation of what he would do including how he would adjust control points in the process. Eventually, he would be allowed to act on real input data, but still would be restricted to going through the motions and explaining what he would do with that real data.

As the teacher gains confidence in the operator's ability to master the procedure, he increases the operator's responsibility. He is permitted to throw a switch, or turn a valve, or influence the process in some way. He may not be allowed to do everything at once but perhaps is permitted to throw the switch but not to affect the process in any other way. The simulation is graduated and the teacher may choose to what degree the operator may affect the process. The user must also have the capability in the android to choose the degree of simulation to be used during debug. Eventually the teacher is satisfied that the procedure has been mastered by the operator—or by the android—and assigns full responsibility.

The android must be capable of assuming responsibility methodically. A responsible procedure is placed in the foreground and becomes part of the real-time system. At that point, no interaction should be permissible between the background and the foreground. The new procedures that will presumably be taught in the future should not be able to influence the proven procedures introduced into the real-time system unless allowed to by the teacher.

The android originally must have been taught a form of organization chart. Not everyone should be permitted access to the background program development features in the system. Just because responsibility can be assigned to the computer does not mean that anybody can assign it. Each potential teacher permitted access to the background features of the system should be assigned his own secret recognition word known only to him and the android. The computer is responsible only to the individual that introduced a particular procedure and who taught the android that procedure. Conflicts of usage can be minimized in this fashion.

There may come a time when something unforeseen occurs and the operator will be asked to account for his actions. The android should retain a diary of what has happened recently for a possible post-mortem call. This diary should give a gross account of the procedures which have been called over the past few moments, the order in which they were called and a rough account of the outputs to the process they made. This permits a post-mortem to be made so that improvement of the procedures can be accomplished.

CONCLUSIONS

Certain features of the android are currently available. Background/foreground processing is offered by a great many suppliers. While it is not as sophisticated as proposed in this paper it is a start in the right direction. Current background processing usually means the ability to compile while foreground programs run real-time. Little emphasis has been placed on the ability to *methodically* link a new

background-produced program into the foreground.

Much of the currently available abilities programmed into a industrial process control computer have been based upon the supplier's experience in specific problems. His background has been in a particular marketing area and he has learned some general approach which seems to have satisfied his previous commitments but which may inherently contain what becomes a preconceived notion when applied to other industrial control areas.

An example is the "generalized scan program" which is intended to gather information from the process and lay it down in core memory. Typically, such a scan program is based upon experience in the petro-chemical field. How useful this approach can be to other industrial processes is an open question. Speaking bluntly, it is an opinionated approach, biased by the experience of the supplier in his early marketing ventures. Much the same can be said about the generalized logging routines which seem to assume the presence of a 30-inch typer. It is very rare when so-called generalized software of the nature of scan and logging programs is accepted readily by the operating personnel who actually have to contend with the process. How many suppliers have thrown up their hands with despair when the software is "modified" by the users! The choice has been, in the past, to modify the software or extensively retrain the operators or ignore the software and start over. But within all these software efforts has been the assumption that there is something basic to all systems having to do with scanning or logging or other operator functions.

Executive routines have been furnished which have a sense of urgency inherent in them. These too are based on prior applications. Most executive systems require the full understanding of all problems to be solved before the first problem can be tackled. The typical industrial process changes with time as the process engineer improves the production equipment or the objectives of the process. Such changes sometimes require a "re-education" of the executive routine.

There are instances in the past of the process control computer being abandoned because the re-education process was too difficult. Perhaps the original programmers had drifted away—as programmers are prone to do—or the original documentation was incomplete, or one of a myriad of problems occurred. Rather than contend with retraining the beast, the computer was downgraded to merely gathering data and printing a log. This is an example of the process evolving without having easy and economical reprogramming features.

Android abilities currently attainable

An extension of the background/foreground scheme would allow for methodical insertion of a debugged program into the real-time system. Some currently available compilers allow for a source language trace of the execution. This can be provided in a compiler intended for industrial control use. Self-documentation is also readily attainable but has normally been limited to computer systems with dedicated peripheral I/O equipment. This feature can be implemented in an industrial control android if the user is willing to dedicate certain I/O devices to the programmer. The self-organization and documentation characteristics desirable in the android require a certain amount of dedicated hardware capability for such a function. Directories of active "programs" in the system, drum mapping and core mapping programs, and other important documentation services require memory and time within the system to maintain up-to-date documentation.

It is important that the android system have an up-to-date library of all source information within the operating system. It must be able to have on call any selected program in order that the user can be assured of what is in the system and understand what the latest version of any program is. One of the weakest links within any user and computer system is the mismatch between the content of the program the user thinks is in the computer system and what the computer programs actually contain. If the computer system can give back only machine language information—a foreign language to the user—it is very difficult for the user to interpret this in order to understand the current situation.

The organization chart for recognition and identification of authority can be readily programmed into a computerized android system. Such a feature requires that the user identify himself prior to any attempt to modify information. Along with the identification capability, the computer system can easily be programmed to incorporate the corresponding sense of discretion required.

It is not difficult to program the sense of time within the computer system. Such statements as delay, or start at 11 o'clock may be incorporated easily within the android system. Simulation is attainable in graduated steps. The first step would be user-supplied inputs to see if the android is solving the problem correctly, then actual field inputs but trapped outputs to determine if the model was realistic. If all input/output calls are channeled to a central monitor, trapping of the actual inputs or outputs and substituted simulation is feasible.

Giving to the android the ability to remember the last few things it did, so that a post-mortem call can be made, is not difficult. Admittedly, it does occupy core memory but with a central monitor structure all activity can be retained up to a certain point. This, of course, is especially useful in the early debugging stages.

A limited degree of self-diagnosis of android problems is also possible. Potential saturation of core or auxiliary memory is not difficult to program and to announce to the user. The android can even detect when it is reaching a saturation of computational time. The android can keep rough track of how much time is spent in useful programs and how much time is spent in training and idle time. Compilers are available in current industrial control computers and eliminate the conversational problem of the user talking to a machine in assembly language. Few compiler offerings provide the ultimate in communication but are a step in the right direction. FORTRAN-based compilers offer the engineer the opportunity to speak in a mathematical language somewhat akin to his; however, none offer the syntax devoted to training the human operator. But it can be done if the syntax used in training can be defined.

Systematic detection of responsibilities can also be programmed now. It is a more difficult thing than merely adding responsibilities because the android may be simultaneously executing that responsibility when the deletion is requested. A deletion of a responsibility can be permitted only when it does not upset current real-time programs. It is not a simple matter of erasing a program previously defined.

Traits that cannot be economically programmed now

Heuristic judgment is currently being programmed in large computers, but is not economically attainable in small computers such as found in industrial control. Self-learning ability would be an ideal trait of the android but unfortunately must await future technological development of the state-of-the-art of programming.

An android that can express lack of understanding is also somewhat in the future. While it is possible and presently attainable for a compiler system to issue diagnostics about obvious misuse of the syntax and misrepresentation of data declaration, anything beyond that is currently unattainable.

The trait of curiosity is also a heuristic ability. The android is only curious about what it has been told to be curious about, and that is not true curiosity; it is only an expression of the programmers will. It may be possible to program the android to express "curiosity" about everything that occurs by causing it to print many messages, but this converts the android into a gabby operator to whom no one listens.

There are some aspects of self-diagnosis which are beyond programming today. Ideally the android would be able to predict that it was going to collapse in the near future or that some small aspect of it was in trouble now or that it was currently having localized problems. Although vendors produce test programs for manufacturing personnel to locate problems during hardware checkout, these programs have rarely been successfully incorporated into a real-time system.

Some projects are currently investigating a form of mobility for the computer. They have attached TV cameras as eyes and pseudo limbs as arms and perhaps some time in the future an economical version of a completely mobile android will occur, but not now. This android must sense the process from a fixed position and must have every input brought to it and every output taken from it by electrical means. It cannot stroll over to a recorder or to a manual station and expect to influence the process.

The android as a student

Programming the digital computer requires a methodical approach. First, the source program is introduced and the android as a student must interpret what has been told to it and digest it in its own time. The android must be able to announce basic misunderstandings about syntax and data declaration. This too can be done by current compilers. Once the source program has been accepted by the android, a debugging procedure is entered. The teacher does not allow the android to assume total responsibility and execute the program in real-time affecting the process. Rather, the teacher approaches debugging in a very cautious way. Artificial inputs are introduced to see what the android's program will do about them. All outputs are trapped, no output is allowed to go through the process. Instead, the output generated by the programs are simulated onto a typewriter so that the teacher may evaluate them. When the teacher is satisfied that synthetic inputs are being satisfactorily handled by the android's program it may permit the android to accept real inputs but still trap the outputs and type them for evaluation. This process proceeds at a pace set by the teacher (the programmer) who evaluates the programs according to a methodical, established debugging pattern. Eventually the teacher will allow the android full responsibility toward this program. The android then assumes responsibility.

Background compilation is quite common these days and, to some degree, introduction of a background-produced program to a real-time foreground program is permitted. However, a methodical debugging of limited yet expanding responsibility has not yet been done. It requires simulation of the inputs and output to the process. This can be done with today's technology of programming. Programs to accomplish that simulation would probably take a considerable amount of memory.

The compilers currently being utilized have the ability to accept data declaration and predefined verbs as a mechanisfor communication. The language requirements of the industrial process user dictate the necessity for the user to define new verbs to the compiler within a limited syntax. He must have the ability to teach the nouns and verbs commonly associated with his process in order that future communications are in a domain familiar to his environment. The associated processing procedure for such verbs as alarm, read, analog point, compare against limit, and set control point must be definable in order that future reference is understandable. This capability is not impossible because a procedural language such as FORTRAN can be used to obtain problem-oriented capability. (DYSTAL and SNOBAL are typical of such language.)

The android approach can be to industrial control computer programming what FORTRAN was to scientific computer programming—a basic system usable across the full breadth of the field while still permitting adjustment by the user to particular applications.

1967 SPRING JOINT COMPUTER CONFERENCE COMMITTEE

Steering Committee

Brian Pollard
Radio Corporation of America
Cherry Hill - Building 204-2
Camden, New Jersey 08101

Donald L. Stevens
Philco Corporation
Fort Washington, Pa.

B. A. Colbert
Price Waterhouse & Company
Independence Mall West
Philadelphia, Pa. 19106

Paul Chinitz
UNIVAC, P. O. Box 500
Bluebell, Pa.

F. M. Hoar
Radio Corporation of America
Cherry Hill - Building 204-1
Camden, New Jersey 08101

R. M. Bennett
IBM Corporation
7 Penn Center
Philadelphia, Pa. 19103

Carl S. Witonsky
IBM Corporation
7 Penn Center
Philadelphia, Pa. 19103

Mary P. Nagle
Radio Corporation of America
Cherry Hill - Building 204-2
Camden, New Jersey 08101

Herman R. Henken
Radio Corporation of America
Cherry Hill - Building 204-1
Camden, New Jersey 08101

Harry M. Haugan
Nat'l Aviation Facilities Experimental Center
Chief Computation Branch
Atlantic City, New Jersey

Richard Ridall
Auerbach Corporation
121 N. Broad Street
Philadelphia, Pa 19107

Donald F. Blumberg
Philco Corporation
515 Pennsylvania Ave.
Fort Washington, Pa.

Robert Snavely
General Electric
Information System and Computer Center
P. O. Box 8555
Philadelphia, Pa

Raymond Dash
Radio Corporation of America
Cherry Hill - Building 204-2
Camden, New Jersey

Henry Hiz
University of Pennsylvania
Philadelphia, Pa 19104

Ned Kornfield
PMC Colleges
Engineering Division
Chester, Pa. 19103

J. Wesley Leas
Control Data Corporation
2621 Van Buren Avenue
Norristown, Pennsylvania

James E. Wolle
General Electric (MSD)
VFSTC
P. O. Box 8555
Philadelphia, Pa.

Mr. John R. Hilligass
513 Inman Terrace
Willow Grove, Pa.

Mr. Cecil C. Hamilton
1535 Hillside Drive
Cherry Hill, N. J.

1967 SJCC Committee (Cont.)

Public Relations

Harry D. Wulforst
UNIVAC
P.O. Box 8100
Philadelphia, Pa. 19104

Ted Swift
News Bureau, General Electric Co.
Barclay Building
Bala Cynwyd, Pa.

Howard H. Babcock
RCA-EDP
Cherry Hill
Camden, N. J. 08101

Lawrence J. Woodward, Jr.
EDP News and Information
RCA-EDP
Cherry Hill
Camden, N. J. 08101

Thomas J. Gradel
EDP News and Information
RCA-EDP
Cherry Hill
Camden, New Jersey 08101

Richard J. Brady
Burroughs Corporation
Defense, Space and Special Systems Group
Paoli, Pa. 19301

Donald G. Dowd
International Operations
Sperry Rand
Sperry Rand Building, Floor 43
1290 Avenue of the Americas
New York, New York 10019

Exhibits

Mr. Richard Zeller
General Electric Company
Information System and Computer Center
P.O. Box 8666
Penn Park Building
Philadelphia, Pa.

Mr. Tristam Coffin
General Electric Company
Information System and Computer Center
P.O. Box 8666
Penn Park Building
Philadelphia, Pa.

Local Arrangements

John Cousins, Vice Chairman
300 - 16th Street, South
Brigantine, New Jersey

Bernard Lewis
402 Monroe Avenue
Linwood, New Jersey

Charles Richardson
111 E. Cheltenham Avenue
Linwood, New Jersey

Edward Boucher
1105 Chelsea Road
Absecon, New Jersey

Edward Dean
20 Kirkland Avenue
Linwood, New Jersey

Louis Allen
6 Highland Court
Linwood, New Jersey

Carl Hazelwood
408 Chestnut Avenue
Northfield, New Jersey

John Hennighan
530 Ridgewood Drive
Northfield, New Jersey

Ray Masters
606 W. Vernon Avenue
Linwood, New Jersey

1967 SJCC Committee (Cont.)

Program

K. Patricia Atkinson
UNIVAC
P.O. Box 8100
Philadelphia, Pa. 19104

Mr. Ed Blumenthal
Burroughs Corporation
Great Valley Laboratory
Paoli, Pa.

Mr. Walter Brunner
Electronic Associates, Inc.
Princeton Computation Center
P.O. Box 582
Princeton, N. J.

Mr. Martin Goetz
Applied Data Research
Route 206 Center
Princeton, N. J.

Margery League
UNIVAC
P.O. Box 8100
Philadelphia, Pa. 19104

Bea M. Roncella
UNIVAC
P.O. Box 8100
Philadelphia, Pa. 19104

Mr. Robert Rossheim
Auerbach Corporation
212 North Broad Street
Philadelphia, Pa. 19107

Mr. Ralph Welken
Philco, Computer Division
3900 Welsh Road
Willow Grove, Pa.

Finance

Mr. James A. Logue
145 Lee Circle
Bryn Mawr, Pa. 19010

Mr. Milton Bauman
268 Byberry Road
Huntingdon Valley, Pa. 19006

Mr. Warren D. Mennig
Radnor Crossing Apts.
Apt. 2-C
278 Iven Avenue
St. Davids, Pa. 19087

Ladies' Program

Miss Ann Gingrich
RCA-EDP
Building 204-2, Cherry Hill
Camden, New Jersey 08101

Barbara Boyle
IBM Corporation
7 Penn Center Plaza
Philadelphia, Pa.

Elizabeth Gunson
IBM Corporation
7 Penn Center Plaza
Philadelphia, Pa.

REVIEWERS, PANELISTS, AND SESSION CHAIRMEN

REVIEWERS

Ben Barnes
George W. Batten, Jr.
W. Breish
J. H. Bryant
B. Bussell
Martin Cohn
M. E. Conway
E. F. Cooley
Barnet Corwin
Peter J. Denning
J. B. Dennis
Frederick D. Dodge
O. Dykstra
P. Eberlein
John F. Egan
Stanley Erdreich
David Finn
S. Fliege
E. B. Fowlkes
Richard Gaudet
A. J. Gehring
M. Gelman
W. Morven Gentleman
T. J. Gleason
I. B. Goldberg
R. W. Hamming
Joseph Hammond
J. K. Hawkins
D. G. Hays
David Hemphill
Betty Holberton
M. Hyman
Bruce B. Johnson
B. J. Karafin
E. A. Kiely
R. A. Kirsch
A. J. Kleinschnitz
C. Lampe
Anthony P. Lannutti
L. R. Lavine
Bernard Lewis
J. A. Lewis
Shen Lin
James MacAleer
Mrs. F. J. MacWilliams
C. L. Mallows
T. J Maloney
G. Marsaglia
Jim McCallister
Mary Lester McCammon
C. B. Moler
Edward Morenoff
H. K. Okamoto
D. Paden
B. Parlett
Thomas Patchell
R. L. Patrick
Warren Plath
S. C. Reed
Paul Richman
Donald Risica
Jerome D. Sable
J. H. Saltzer
F. J. Sansom
Parlan Semple, Jr.
Rabah Shahbender
H. A. Simon
James Smith
S. S. Soo
Robert Stevens
I. Tarnove
Leroy N. Tempelton
John VanderVeer
J. Varah
David J. Waks
John D. Williams
R. L. Wigington
P. Wynn
Hideo Yamada
Tseute Yang
E. E. Zajac

PANELISTS

Mr. Gene Amdahl
Mr. James D. Babcock
Mr. Ben Barnes
Mr. Robert S. Barton
Mr. R. W. Bemer
Mr. Lawrence I. Boonin
Mr. Richard M. Brown
Dr. Lawrence E. Burkhart
Dr. Sylvia Charp
Mr. Carlos H. Christensen
Mr. A. B. Clymer
Dr. Myron L. Corrin
Mr. J. B. Dennis
Mr. Cecil B. Dotson
Mr. John F. Egan
Mr. Bob O. Evans
Mr. David Finn
Mr. Donald Frush
Mr. Richard H. Fuller
Col. Paul Galentine
Dr. Edward L. Glaser
Mr. Julian Green
Mr. Dale Gunderson
Mr. Mark Halpren
Mr. Dennis Hamilton
Dr. Duncan Hansen
Miss Betty Holberton
Prof. Robert Howe
Mr. Fred Ihrer
Mr. P. Z. Ingerman
Mr. Bruce B. Johnson
Mr. Richard C. Jones
Mr. Barry J. Karafin
Mr. A. I. Katz
Mr. W. H. Kautz
Dr. Robert Kohr
Dr. Ladis E. Kovach
Mr. Howard S. Krasnow
Dr. D. Kuck
Mr. Peter Landin
Prof. Leon Lapidus
Mr. L. E. Lavine
Mr. Robert N. Linebarger
Mr. W. R. Lonergan
Mr. Brad Mackenzie

PANELISTS—Continued

Dr. Mary McCammon
Mr. M. E. McCoy
Dr. Max Palevsky
Mr. David L. Parnas
Mr. Jerome H. Saltzer
Miss Jean E. Sammet
Mr. Joseph Sciulli
Mr. Parlan Semple, Jr.
Mr. Rabah Shahbender
Mr. Daniel L. Slotnick
Dr. Avrum Soudack
Mr. Paul T. Veillette
Dr. Jacques Vidal
Mr. David J. Waks
Dr. H. S. Witsenhausen
Mr. George P. West

SESSION CHAIRMEN

Mr. Lowell S. Bensky
Mr. E. I. Blumenthal
Mr. Theodore H. Bonn
Dr. J. W. Carr, III
Mr. K. P. Clancy
Mr. Wesley A. Clark
Mr. Gregory M. Dillon
Mr. Paul Dixon
Mr. John M. Evans
Mr. Martin Goetz
Dr. Saul Gorn
Mr. T. J. B. Hannom
Mr. Harry M. Haugan
Mr. Gerhard L. Hollander
Mr. Anatol Holt
Dr. Grace Murray Hopper
Mr. Morton C. Jacobs
Mr. Ray Lawrence
Mr. James L. Maddox
Dr. Robert McNaughton
Mr. Baker A. Mitchell, Jr.
Mr. Gordon Mitchell
Mr. I. D. Nehama
Mr. Robert Rossheim
Mr. L. Scholten
Mr. John C. Strauss
Dr. Joseph F. Traub
Mr. Richard E. Utman
Mr. David VanMeter
Mr. Robert Vichnevetsky
Dr. Willis Ware
Mr. Ralph L. Welken
Mr. Stephen E. Wright

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

345 E. 47th Street, New York, New York 10017

Officers and Board of Directors

President

DR. BRUCE GILCHRIST*
IBM Corporation
Data Processing Division
112 East Post Road
White Plains, New York 10601

Secretary

MR. MAUGHAN S. MASON
Dept. 210
IBM Corporation—FSD
P. O. Box 1250
Huntsville, Alabama 35805

Vice-President

MR. PAUL ARMER*
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Treasurer

MR. WILLIAM D. ROWE *
Sylvania Electronics Systems
189 B. Street
Needham Heights, Massachusetts

ACM Directors

DR. ANTHONY G. OETTINGER
Harvard Computation Laboratory
Cambridge, Massachusetts 02138

DR. ROBERT W. RECTOR*
Informatics, Inc.
5430 Van Nuys Boulevard
Sherman Oaks, California 91401

MR. J. D. MADDEN
ACM Headquarters
211 East 43rd Street
New York, New York 10017

DR. WALTER HOFFMAN
Computing Center
Wayne State University
Detroit, Michigan 48202

IEEE Directors

MR. SAMUEL LEVINE
Bunker-Ramo Corporation
445 Fairfield Avenue
Stamford, Connecticut 06904

MR. KEITH W. UNCAPHER
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

DR. T. J. WILLIAMS
Control & Information Systems Laboratory
Purdue University
Lafayette, Indiana 47907

DR. R. I. TANAKA*
California Computer Products
305 North Muller Street
Anaheim, California 92803

Simulation Councils Director

MR. JOHN E. SHERMAN*
Lockheed Missiles & Space Co.
D-59-10, B-151
P. O. Box 504
Sunnyvale, California 94088

American Documentation Institute Director

MR. HAROLD BORKO
System Development Corporation
2500 Colorado Avenue
Santa Monica, California 90406

Association for Machine Translation and Computational Linguistics-Observer

DR. DAVID G. HAYS
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Executive Secretary

MR. H. G. ASMUS
AFIPS Headquarters
345 E. 47th Street
New York, New York 10017

* Executive Committee

AFIPS Committee Chairmen

Abstracting

DR. DAVID G. HAYS
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Admissions

MR. WALTER L. ANDERSON
General Kinetics, Inc.
2611 Shirlington Road
Arlington, Virginia 22206

Awards

DR. ARNOLD A. COHEN
UNIVAC
2276 Highcrest Drive
Roseville, Minnesota 55113

Conference

DR. MORTON M. ASTRAHAN
IBM Corporation—ASDD
P. O. Box 66
Los Gatos, California 95030

Constitution and By-Laws

MR. MAUGHAN S. MASON
Dept. 210
IBM Corporation—FSD
P. O. Box 1250
Huntsville, Alabama 35805

Education

DR. MELVIN SHADER
IBM Corporation
100 Westchester Avenue
Harrison, New York 10528

Finance

MR. WALTER M. CARLSON
IBM Corporation
Old Orchard Road
Armonk, New York 10504

Harry Goode Memorial Award

DR. WILLIS H. WARE
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

IFIP Congress 68

DR. DONALD L. THOMSEN, JR.
IBM Corporation
Old Orchard Road
Armonk, New York 10504

Government Advisory

DR. HARRY HUSKEY
Massachusetts Institute of Technology
Cambridge, Massachusetts

International Relations

DR. EDWIN L. HARDER
1204 Milton Avenue
Pittsburgh, Pennsylvania 15218

Planning

DR. JACK MOSHMAN
EBS Management Consultants, Inc.
1625 I Street, N.W.
Washington, D. C. 20006

Public Relations

MR. ISAAC J. SELIGSOHN
IBM Corporation
Old Orchard Road
Armonk, New York 10504

Publications

MR. STANLEY ROGERS
P. O. Box R
Del Mar, California 92014

Publications Task Force

MR. SOL ROSENTHAL
HQ USAF
AFA DAC
Pentagon
Washington, D. C. 20330

Social Implications of Information Processing Technology

MR. HERBERT KOLLER
EBS Management Consultants, Inc.
1625 I Street, N.W.
Washington, D. C. 20006

Technical Program

MR. JACK ROSEMAN
2313 Coleridge Drive
Silver Spring, Maryland 20910

Newsletter

MR. DONALD B. HOUGHTON, 15-W
Westinghouse Electric Corporation
3 Gateway Center, Box 2278
Pittsburgh, Pennsylvania 15230

COSATI Liaison

MR. GERHARD L. HOLLANDER
Hollander Associates
P. O. Box 2276
Fullerton, California 92633

Consultant

MR. HARLAN E. ANDERSON
Time, Inc.
New York, New York

JCC General Chairmen

1967 FJCC

MR. LINDER C. HOBBS
Hobbs Associates, Inc.
P. O. Box 686
Corona del Mar, California 92625

1968 SJCC

DR. A. S. HOAGLAND
I.B.M. Research Center
Yorktown Heights, New York

1967 SJCC LIST OF EXHIBITORS

Adage, Inc.
Adams Associates, Inc.
Addison-Wesley Publishing Co., Inc.
Addressograph Multigraph Corp.
Advanced Computer Techniques Corp.
Amp. Inc.
Ampex Corp.
Anelex Corp.
Applied Data Research, Inc.
Applied Dynamics, Inc.
Association for Computing Machinery
Auerbach Corporation
Auto-trol Corp.

Baldwin Kongsberg Co.
Bell System
Benson-Lehner Corp.
Beta Instrument Corp.
Bolt, Beranek and Newman, Inc.
Brogan Assoc., Inc.
Burroughs Corp.-ECD
Burroughs Corp.-Corp. Communications

California Computer Products, Inc.
Calma Co.
Comcor, Inc.
Compat Corp.
Computor Design Publishing Corp.
Computer Sciences Corp.
Computers and Automation
Computer Test Corporation
Computer Usage Co., Inc.
Concord Control, Inc.
Consolidated Electrodynamics Corp.
Control Data Corp.
Corning Glass Works
Cybetronics, Inc.

Dartex
Datamation
Data Pathing Inc.
Data Processing Magazine
Decision Control, Inc.
Dial-Data, Inc.
DI/AN Controls, Inc
Digi Data Corp.
Digital Devices, Inc.
Digital Equipment Corp.
Digitronics Corp.

Electronic Associates, Inc.
Electronic Memories, Inc.

Fabri-Tek Inc.
Ferranti-Packard Electric Limited
Ferroxcube Corp. of America

General Computors, Inc.
General Kinetics Inc.
General Precision/Librascope
Geo Space Computer Div.
GPS Instrument Co., Inc.

Hewlett-Packard Datamec Div.)
Hewlett-Packard Dymec Div.)
Holt, Rinehart and Winston, Inc.
Honeywell, Computer Control Div.
Honig Time Sharing Association, Inc.
Houston Omnigraphic Corp.

Indiana General Corp.
IBM Corp. -DP Div.
Interdata
Interstate Electronics Corp.
Invac Corp.
ITT-Ind. Prod. Div.

Kennedy Co.
Kleinschmidt, Div. of SCM

Lancer Electronics Corp.
Lenkurt Electric Co., Inc.
Litton Electronics Business Systems
Litton Industries, DATALOG Div.
Lockheed Electronics Co./A.I.P. Div.

Magne-Head, Div. of Gen'l Instr. Corp.
McGraw-Hill Book Co.
Micro Switch Div. of Honeywell
Midwestern Instruments/Telex
3M Co. -Mag. Products Div.)
3M Co. -Microfilm Products Div.)
3M Co. -Revere-Mincom Div.)
3M Co. -Visual Products Div.)
3M Co. -Reflective Products Div.)
Monitor Systems, Inc.
Mosaic Fabrications, Inc.

NCR

List of Exhibitors (Cont.)

Patwin Electronics
Precision Instrument Co.
Prentice-Hall, Inc.

RCA-EC&D Div.
RCA-EDP Div.
Raytheon Co.
Raytheon Computer
Redcor Corp.
Rixon Electronics Inc.

Sanders Assoc., Inc.
Scientific Data Systems, Inc.
Soroban Engineering, Inc.
Spartan Books
Sylvania Electric Products Inc.
Systems Engineering Labs, Inc.

Tally Corp.
Tasker Industries
Teletype Corp.
Texas Instruments Inc. (IPG)
Thompson Book Co.
Transistor Electronics Corp.

UNIVAC
University of Pittsburgh
U. S. Magnetic Tape Co.

Vermont Research Corp.

Western Union
John Wiley & Sons, Inc.

Xerox Corp.

Zeltex, Inc.

AUTHOR INDEX

Adams, E. N., 419
Aleksander, I., 707
Allen, J., 623
Amdahl, G. M., 483
Anderson, M. D., 133
Andrews, K. B., 169
Anne, A., 393
Aschenbrenner, R. A., 81
Auroux, A., 547
Babcock, J. D., 301
Ball, W. E., 377
Bandat, R. S., 457
Barron, D. W., 163
Bartram, P. R., 635
Bedrosian, S. D., 157
Bellino, J., 547
Bequaert, F. C., 571
Blum, A. S., 365
Bolliet, L., 547
Brevik, D. J., 783
Brian, D., 623
Burkhart, L. E., 487
Cain, J. T., 757
Champine, G. A., 541
Chaney, T. J., 365
Chang, G. D., 691
Chow, W. F., 507
Citron, J. P., 553
Clark, W. A., 335, 337
Cody, W. J., 305
Coggan, B., 645
Cohen, L. J., 671
Conn, R. W., 103
Cook, P. A. C., 779
Corrin, M. L., 487
Crane, B. A., 517
Crocker, S. D., 645
Dammkoehler, R. A., 371
Denning, P. J., 9
Devine, J. L., Jr., 257
Dixon, P. J., 185
Emmons, S. P., 121
Ernst, G. W., 583
Ershov, A. P., 577
Estrin G., 645
Evans, D. C., 23
Farrand, W. A., 239
Feldman, G., 623
Flynn, M., 81
Fraser, A. G., 163
Fuller, R. H., 471
Gaines, B. R., 149
Gardiner, J. T., 245
Gelman, M., 413
Gibson, D. H., 75
Gielow, K. R., 657
Gilbert, P., 447
Ginsberg, A. S., 441
Glaser, E. L., 303
Gold, L. D., 253, 273
Goldstein, A. J., 325
Gorn, S., 213
Guida, P. M., 273
Gupta, S. C., 133
Halstead, M. H., 657
Hammer, C., 331
Hartley, D. F., 163
Hollander, G. L., 463
Holmes, W. S., 265
Hopkins, D., 645
Horsfall, R. B., 239
Humphrey, T. A., 719
Hurwitz, A., 553
Igarashi, R., 499
Ingalls, V. W., 125
Iwata, J., 141
Jacobowitz, H., 761
Jallen, G. A., 729
Jauvtis, H. I., 491
Kaiser, E. P., 531
Kapps, C. A., 677
Katz, A. I., 487
Keller, E. R., II, 157
Kim, J. C., 531
Kodzuhin, G. I., 577
Kohavi, Z., 713
Kohr, R., 487
Kovach, L. E., 487
Laane, R. R., 517
Landy, B., 163
Lavallee, P., 713
Leclerc, J. Y., 23
Leroy, H., 663
Libby, R. L., 227
Liebowitz, B. H., 51
Liu, C. L., 691
Lombardo, J. M., 771
Loudon, R. G., 257
Love, H. H., Jr., 87
MacDougall, M. H., 735
Makapov, G. P., 577
Marcum, N. E., 239
Marcus, R. S., 227
Marino, M. J., 491
Markowitz, H. M., 441
Matula, D. W., 311
McCarthy, J., 623
McCoy, E. K., 433
McLean, J. B., 175
McLellan, W. G., 447
McNamee, L. P., 757
Meeker, R. J., 525
Merritt, C. A., 429
Mickle, M. H., 757
Miura, T., 141
Moler, C. B., 321
Molnar, C. E., 393
Moore, S. W., 273
Moore, W. H., Jr., 525
Moreno, A. H., 253
Morenoff, E., 175
Neblett, J. B., 783
Nechepurenko, M. I., 577
Needham, R. M., 163
Neil, G., 45
Nelson, D. B., 169
Newell, A., 583
Oldfather, P. M., 441
Olsen, R. E., 365
O'Neill, R. W., 611
Ornstein, S. M., 337, 393
Patt, Y. N., 699
Peters, B., 283
Petersen, H. E., 291
Phillips, C. E., 265
Pick, R. A., 169
Piligian, M. S., 61
Pokorney, J. L., 61
Pottosin, I. V., 577
Prell, E. M., 765
Puterbaugh, W. H., 121
Ramamoorthy, C. V., 743
Ratynski, M. V., 33
Reiter, A., 1
Robinson, G. A., 81
Sable, J. D., 185
Savitt, D. A., 87
Schneider, V. B., 685
Searle, L. V., 45
Shure, G. H., 525
Slotnick, D. L., 477
Smith, W. A., Jr., 425
Soudack, A., 487
Spain, R. J., 491
Spandorfer, L. M., 507
Stallard, L. B., 227
Stanga, D. C., 67
Steil, G. P., Jr., 199
Stucki, M. J., 357
Thorlin, J. F., 641
Troop, R. E., 87
Tsuda, J., 141
Turn, R., 291
Uber, G. T., 657
Van Dam, A., 601
Walter, C. J., 107
Ware, W. H., 279, 287
Welch, A. J., 257
West, G. P., 467
Wexelblat, R. L., 559
Wilkins, R. L., 457
Williams, W. D., 239
Williams, W. D., 635
Wodtke, K., 403
Yaita, T., 499
Yeaton, J. B., 553