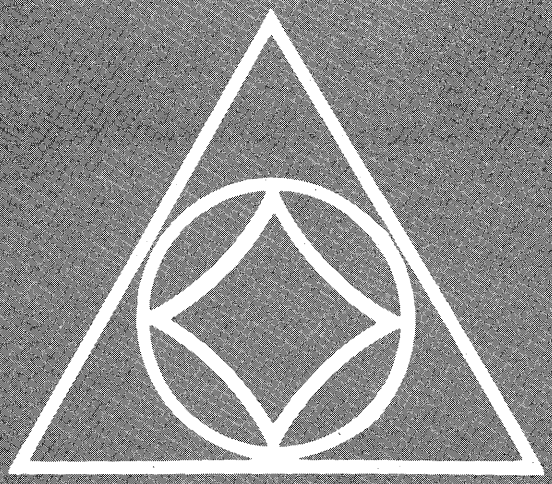


AFIPS

CONFERENCE PROCEEDINGS

VOLUME 36

1970

SPRING JOINT COMPUTER CONFERENCE

May 5-7, 1970
Atlantic City, New Jersey

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1970 Spring Joint Computer Conference Committee or the American Federation of Information Processing Societies.

Library of Congress Catalog Card Number 55–44701

AFIPS PRESS
210 Summit Avenue
Montvale, New Jersey 07645

Printed in the United States of America

CONTENTS

Editor's Note:

Due to the recent embargo of mail, several papers went to press without author and/or proofreader corrections.

An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources

by J. BOUKNIGHT and K. KELLEY

University of Illinois
Urbana, Illinois

INTRODUCTION

In the years since the introduction of SKETCHPAD an increasing number of graphics systems for line drawing have been developed. Software packages are now available to do such things as picture definition, rotation and translation of picture data, and production of animated movies and microfilm. Automatic windowing, three-dimensional figures, depth cueing by intensity, and even stereo line drawing are now feasible and in some cases, available in hardware.

Even with all these capabilities, however, representation of three-dimensional data is not quite satisfactory. Representing a solid object by lines which define its edges leads to the computer generated unreality of being able to see through solid objects. In recent years, research centered around means for computer graphical display of structural figures and data has begun to move from display of "wire-frame" structures where the "wires" represent the edges of the surfaces of the structures, to the display of structures using surface definition techniques to enhance the three-dimensional appearance of the final result. Several efforts have been concentrated on producing graphical output which is similar to the half-tone commercial printing process.

The work of Evans, et al., at the University of Utah[1] established the feasibility of using a computer to produce half-tone images. Their algorithm processes structures whose surfaces are made up of planar triangles. The algorithm employs a raster scan and examines crossing points of the boundaries of the triangles by the scanning ray. A significant feature of their method is that the increase in computing time is linear as the resolution of the picture increases.

John Warnock's algorithm for half-tone picture representation employs a different technique.[2] He divides the scene recursively into quarters until all detail in a given square is known or the smallest size square is reached. The result is a set of "key squares", that is intensity change points, along the visible edges in the scene. The time required for this algorithm varies linearly as the total length of the visible edges in the picture, but varies also as the square of the raster size. An important feature of the Warnock algorithm is that it handles the occurrence of the intersection of two planes without having to precalculate the line of intersection.

At the General Electric Electronics Research Laboratory in Syracuse, a system which combines both hardware and software to produce color half-tone images in real time has been developed for NASA as a simulator for rendezvous and docking training. This device can hold up a 600 $\times$ 600 raster point picture of up to 240 edges, in color, and change the picture as quickly as the beam scans the screen.[3]

The work of the computer group at the Coordinated Science Laboratory began as an effort to add some realism to line drawings of structures being generated by R. Resch, who, while working in the laboratory, was also a member of the faculty of the Department of Architecture. Through his acquaintance with J. Warnock, we were able to implement a version of the Warnock algorithm which operates on the CDC 1604. After several revisions of the implementation and some fine tuning of the CRT display hardware, black and white half-tone images of the Resch structures were exhibited at the Computer Graphics Conference at the University of Illinois in April of 1969.

In discussions with J. Warnock and Robert Schumacher of General Electric, we envisioned a hidden surface algorithm using a scanline technique combining the recursiveness of the Warnock algorithm with the hardware techniques used in the NASA simulator.

These discussions were the impetus for the development of the LINESCAN algorithm.[4] It was for this implementation that laboratory engineers added a raster scan operation to the display hardware. As a result, the algorithm does not have to provide the scope with an item of data for each and every point. Only the location of an intensity change on the scanline and the new magnitude of the intensity are needed. In addition, the display hardware was modified to allow 256 levels of intensity under program control.

Neither of the previously mentioned algorithms for half-tone images represented the picture with a light source located away from the observer position, although both regard it as a next step in development. Moving the light source away from the observer position presents the problem of cast shadows. Arthur Appel's work approaches the shadow problem and the hidden line or surface problem simultaneously.[5] His algorithm also scans the picture description in a linescan manner. The question of which parts of which surfaces are visible is answered by a technique called "quantitative invisibility[6]". His structures are composed of planar polygonal surfaces. Appel also includes the ability to handle multiple illumination sources and the shadows cast due to those sources. His shadow boundaries are computed by projecting points incrementally along the edge of a shading polygon to the surface which will be shaded.

The work of the Computer Group at the Coordinated Science Laboratory to move the light away from the observer began shortly after the completion of the LINESCAN algorithm. Augmentation of the original LINESCAN method with a dual scan for shadows cast upon the surfaces presents two questions.

First, the number of projected shadows to be calculated must be kept to a minimum; but the technique for narrowing the set of polygon pairs must be simple. For a single illumination source, we are constrained by the fact that for n polygons, there are $n(n-1)$ pairs to be considered. The method chosen to narrow the set of shadow casting and receiving polygons was to project the polygons onto a sphere centered at the light source and make some gross comparisons of maximum and minimum Euclidean coordinates of the points so projected.[7] The transformation to the sphere was so devised that no trigonometric functions or square roots were used. The comparisons used are not intended to discard all pairs that do not cause shadows. The point is to discard as possible shadow pairs all cases in which it is obvious that a shadow is not cast on one by the other. Some nonshadow-producing pairs do slip through the first set of tests; this is allowed because the second set of tests can check these cases with less overall programming effort and execution time.

The second question which the algorithm answers is how to handle the most prevalent situation of shadows cast by one polygon only partially falling on another polygon. It is not necessary to compute the boundaries of the intersection of a polygon with a shadow cast upon its plane. The decision to reduce intensity as the scan enters the shadowed portion of a polygon is left to the final picture-producing stage of the process. Also in the present version no computation is wasted to see if the cast shadow is visible to an observer. Shadows are output with tags to tell which polygon is being shadowed and which polygon is casting the shadow. The final step of the process responds to shadow information by making appropriate intensity changes only in the case that the shadowed polygon is the same as the current visible polygon.

THE LINESCAN ALGORITHM AND ITS ADAPTATION FOR SOLVING THE SHADOW PROBLEM

The LINESCAN algorithm presents itself as a likely candidate for extension to a system for solving the shadow problem in half-tone image processing because of its speed of operation and because it is directly suited to processing a shadow-space which is structured in the same manner as the associated three-space polygonal surface structures. Shadows cast by one polygon onto another by point illumination sources are them-

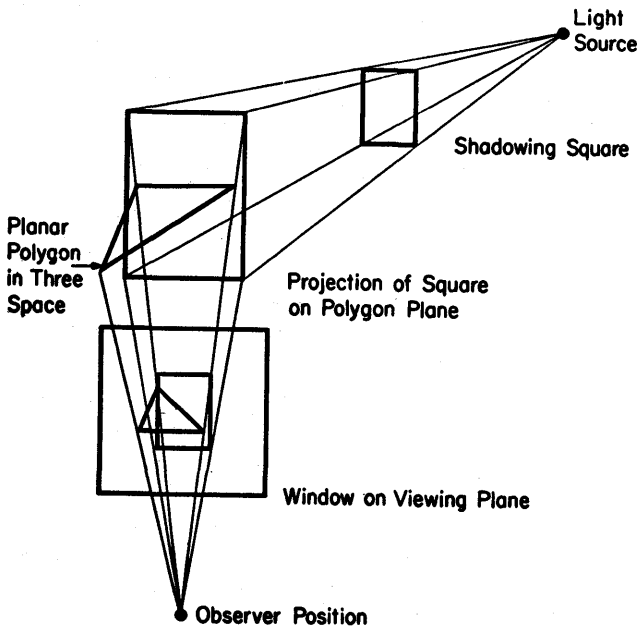


Figure 1—Shadow and object projections

selves polygons in the same three-space. The resulting shadow three-space can be projected onto the viewing plane in the same way as the original three-space structure (see Figure 1). Thus, the extension of the LINESCAN algorithm involves only the addition of a second scanning process for keeping track of shadows on each scanline.

A brief description at this point will serve to orient the reader to the actual mechanisms of the LINESCAN algorithm. The LINESCAN algorithm processes a graphical image into a half-tone final image from two data sets derived from the three-space structure: (1) the set of all plane equation coefficients for the polygonal surfaces of the structure and (2) the perspective projections of the edges of the surfaces on the viewing plane.

The construction of the final half-tone image is done in a television-like manner where a CRT beam scans across the image line-by-line and exposes a raster of points. As the beam moves across the scanline, the intersection points on the scanline corresponding to the viewable edges of the original structure will dictate changes in the tone (intensity) of the scanning beam from that point to the next intersection. These intersection points, which are output to the final image producing routine as "key squares", are the primary output data produced by the LINESCAN algorithm.

"Key squares" are generated in two types of locations on the viewing plane during a linescan operation. The primary type of location is the intersection of the current scanline with the projection of an edge of the three-space structure. This location will cause a "key square" to be produced only if the intersection is visible to the observer.

The second type of location on the viewing plane which can cause a "key square" to be produced is the intersection of an "implicitly defined line" and the current scanline. An "implicitly defined line" is the projection on the viewing plane of the intersection of two or more polygons in three-space. Polygon intersections are allowed in the theoretical world of the computer even though they violate the law of Nature that only one object may occupy a given amount of space. Because of the implicit nature of these intersections, special operations must be performed to detect and process them to produce the correct final result.

For any given scanline, a linked list is present containing all intersections of projected edges with that scanline ordered in the direction of the scanning movement. The LINESCAN algorithm moves from intersection to intersection keeping track of which polygon projections are entered and which ones are exited by the scanning ray. At each intersection, a "depth sort" is performed on those polygons being pierced by the scanning ray to find the closest or visible polygon at that intersection on the scanline.

The decision to produce a "key square" at a given intersection point is based primarily upon the relation of the depth of the edge associated with the point and the visible polygon determined by the "depth sort" at the intersection. If the edge is visible, then consideration is given to whether a polygon projection will be entered as this edge is crossed or whether it will be exited. For the entering case, the "key square" will denote the new polygon for control of the CRT scanning beam at that intersection. For the exiting case, the "depth sort" polygon will be denoted.

Two special problems arise concerning the output of multiple "key squares" for a given point on the final image raster. The first requires that constant checking be performed to see if the integer value of successive intersection points on the scanline are equal. If this occurs, any "key square" action which would be taken for any given intersection in the group will be deferred until the last member of the group has been processed. Thus, only one "key square" will actually be produced. The polygon to be denoted by the resulting "key square" may change from the beginning of the group to the end; but in any case, the last visible polygon will control the result.

The second special case of close intersections occurs when coincident edges occur in the specification of the three-space structure. Performing a "depth sort" on the associated polygons at the common intersection would normally fail because their depths would be the same. Determination of the visible polygon of the group is performed by actually moving the scanning ray a small increment forward in the scanning direction and computing the "depth sort" at that point. This will yield the polygon which will be visible just after the scanning ray leaves the coincident intersection point.

"Implicitly defined lines" are detected when the visible polygon denoted at two successive intersection points is different. The procedure used in searching for the projected intersection involves finding which polygons are intersecting and using their plane equation coefficients to calculate their intersection's projection and its intersection with the scanline. An iterative procedure is used in order to detect the possibility of multiple pairs of intersecting polygons which might yield more than one "implicitly defined line". "Key squares" will be produced for the calculated intersection points on the scanline subject to the same constraints about multiple "key squares" for the same raster points in the final image.

The extended version of the LINESCAN algorithm for solving the shadow problem includes two scanning

operations. The primary scanning movement is the original scan operation where the three-space structure data is processed to provide the final image structure. The additional or secondary scanning operation processes, in a parallel manner, the shadow three-space structure to produce data which will be combined with the primary scan data to form the scanline intensity data for the final image. The data output from the secondary scanning operation, which we call "shadow key squares", affects the intensity patterns of the final image only. Only the primary scan data defines the structure.

In order to keep the changes in the LINESCAN algorithm to a minimum and for any changes made to have minimum influence in computation speed of the implementation, it was decided that as much of the processing operations for the final image as was possible would be shifted to the final output routine from the LINESCAN routine. This was because the original final output routine, PIXSCANR, had been input-output bound during the processing of the "key squares" data file from the LINESCAN routine. The additional operation impressed on the new version of PIXSCANR was the keeping track of which shadow polygon projections were being pierced by the scanning ray at any point in the processing of a scanline. Thus, the only change to the LINESCAN algorithm involved the addition of the secondary scan which simply detected the crossings of the scanline by projected edges of the shadow three-space structure and issued "shadow key squares" at every occurrence.

The dichotomy imposed on the shadow processing responsibilities between the LINESCAN and PIXSCANR routines has an additional advantage. There will always be the possibility of cast shadows falling outside the visible portions of the associated polygons. Additionally, some polygonal surfaces of the original surface will not appear in the final image and therefore, neither will their shadows. We shall see in our discussion of shadow pair detection that it is most economical to allow shadow projections of these kinds to be processed in the same manner as all other shadow projections. Their data items will be passed on to the PIXSCANR routine where their occurrence will be duly noted. No effect will be registered on the final image, however, since the associated three-space surface polygon will not appear in the final image or at least not in conjunction with the projection of the extraneous shadow.

Once the mechanism for producing the proper final image of the half-tone presentation was established, it remained to develop the proper procedure for comparing all possible pairs of polygons with respect to the illumination source, and in an economically feasible manner, discard as many extraneous shadow pairings as possible. Economy of computation speed relative to total scanline processing time was the main concern.

SHADOW DETECTION

The primary task to be accomplished in shadow detection is not so much the actual projection of shadows as it is the elimination of the need for calculating projections and storing shadow polygons unnecessarily. The number of possible shadows cast is equal to the number of possible pairs of polygons in the structure. Since this number increases rapidly as the complexity of the structure increases, it is extremely important to be able to identify useful shadow pairs with a minimum of computation and to store this information in a compact form.

The shadow pairs are stored in a chained list, with subchains linking all polygons that may shadow a given polygon. The procedure for narrowing the set of all possible pairs of polygons to a near minimal set of shadow producing pairs consists of two distinct steps. In the first the polygons are projected onto a sphere centered at the light source and are checked in an approximate fashion for interference with respect to the light source. In the second step, pairs of polygons which seem to occlude one another are further examined to determine which polygons may shadow the others.

The light sphere projection is a device for culling out certain pairs of polygons which can in no way interfere with respect to a given light source. It is only a gross test, intended to ease the burden of computing projections of one polygon onto the plane of another. The test throws out polygon pairs only if it

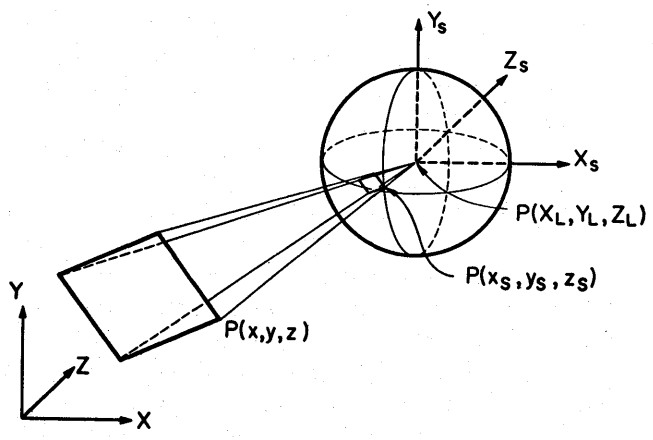


Figure 2—Projection of polygon on sphere

is obvious that no interference takes place. The light sphere projection is in no way used to compute intersections of polygon projections on the sphere, nor is it used to compute intersections of shadow polygons with the polygons being shadowed.

Every vertex of the three-space structure has to be projected onto the sphere centered at the light source in order to make initial interference tests. The sheer magnitude of the number of operations necessary restricts the projection in a number of ways. Namely, it would be preferred if the job could be done without computing any trigonometric functions and if possible, without computing very many square roots, since each of these requires much computer time. The projection is given by:

$$X_s = \frac{\text{sgn}(X_l) * X_l^2 * K^2}{\text{DELTA}}$$

$$Y_s = \frac{\text{sgn}(Y_l) * Y_l^2 * K^2}{\text{DELTA}}$$

$$Z_s = \frac{\text{sgn}(Z_l) * Z_l^2 * K^2}{\text{DELTA}}$$

where X_l, Y_l, Z_l are the coordinates of a point with respect to an origin at the light source, and DELTA is the square of the distance from the point to the light source (see Figure 2). This transformation to the light sphere is a composite of four transformations which are done algebraically to arrive at the final transformation:

(1) transform the points to an Euclidean 3-space with origin at the point of light;
(2) transform these coordinates to polar coordinates;
(3) map these points to the sphere by setting ρ to a constant for each point; and
(4) transform the points on the sphere back into the Euclidean 3-space with origin at the light source.

The algebraic derivation of these transforms yields a final form that involves some square roots in the numerator and denominator. However, since only the relative magnitude is used in the comparison operations, these results are all squared; and the sign is preserved, yielding the final transformation.

In order to use the transformed points to determine which polygons interfere with each other with respect to the light source, the maximum and minimum of the X_s, Y_s, Z_s, and DELTA values are saved for each polygon, in addition to the transformed points. Also the coefficients of the equations of the planes in the light source space are computed and saved for possible shadow computation.

The first check made for each polygon is to see if it is self-shadowed, that is, to see if the observer and light source are on opposite sides of the plane of the polygon. The procedure is to substitute both the light source point and the observer point into the equation of the plane. If the two results have different signs, the polygon is self-shadowed and no shadows cast on it will be computed. However, shadows cast by the self-shadowed polygon must still be considered.

If a polygon is not self-shadowed, then it is compared to each remaining polygon in the list to see if it is obvious that interference does not occur. The criterion is as follows:

> For all pairs of polygons P_i and P_j, if the points transformed to the sphere are separated in X_s, Y_s, or Z_s, then the polygons do not interfere with each other with respect to the light source.

This criterion amounts to simply examining the orthographic projection of the points on the sphere onto the coordinate planes and looking for separation by comparing maximums and minimums in each direction. In the event that the projection of a polygon is so oriented on the sphere as to wrap around a coordinate axis, then the maximum or minimum in some direction does not occur at a vertex. In this case, for the purpose of this comparison, the associated maximum or minimum is replaced by the absolute maximum or minimum coordinate value on the sphere.

When a pair of polygons are not separated enough for this test to detect the separation, then the maximum and minimum distances to the light source are compared. If the maximum distance of the vertices of polygon I from the light source is less than the minimum such distance on polygon J, then it is clear that polygon I may cast a shadow on polygon J but not vice versa.

The tests of projections on the light source sphere eliminates many possible shadow pairs and thus reduces the total amount of storage and computing time required. This set of tests, however, fails to eliminate certain other shadow pairings which will not affect the final image. Among these are shadow pairings of polygons with common vertices and of polygons which completely overlap on the sphere. In neither case is there clear separation on the light source sphere. In the latter case the sizes of the polygons may be so disparate as to nullify the usefulness of vertex distance comparisons.

Figure 3 illustrates one of the cases in which the projection of points of polygon I onto the plane of polygon J indicates the presence of a shadow, while the shadow cast in no way falls within the bounds of polygon J. Since there is no separation between the two polygons, both possible shadow pairings would be

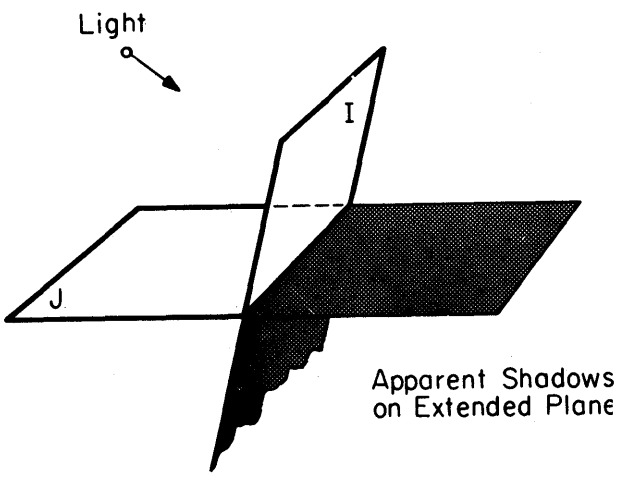


Figure 3—Orientation of polygon pairs—Case 1

noted, and both would be computed, but neither would be present in the final picture.

We can eliminate this case and several others like it (see Figure 4) by defining and appropriately testing two relations:

I → J: "The planar polygon I is entirely on one side of the plane of planar polygon J."

I s J: "Each point of planar polygon I lies between the light source and the plane of planar polygon J."

In the case if Figure 3 for example, we see that I → J, J → I, I s J, and J s I are all true, and therefore, the shadows are cast only upon the extended planes of the two polygons. As a result, neither shadow pairing is added to the list of possible shadows.

All of the decisions about possible shadow pairs are computed in advance of the start of the LINESCAN operation. As the LINESCAN operates, it has a linked list for each polygon of all polygons which may cast a shadow upon it. It is at the point where the first line of a polygon is processed that the shadows cast upon it are computed and stored in a list. When the polygon is no longer active, we purge the shadow information from the list. Thus, shadow information is calculated only when it is first needed and discarded when the need for it ceases.

Shadows are computed by projecting the vertices of one polygon onto the plane of another. The parametric form of the equation of a line is used to calculate this projection. Given two points $P_1(X_1, Y_1, Z_1)$ and

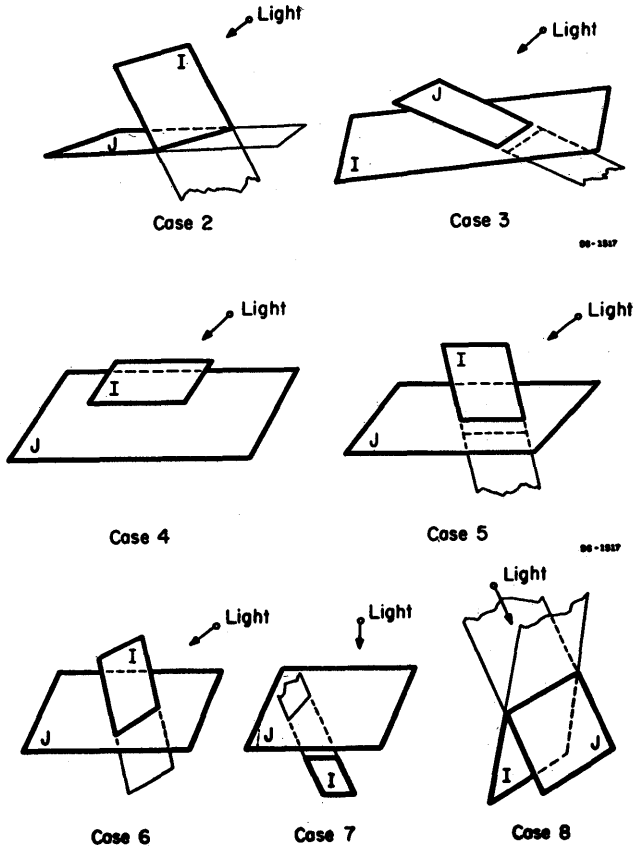


Figure 4—Polygon orientations

$P_2(X_2, Y_2, Z_2)$, the set of points $P(X, Y, Z)$ that lie on a line joining P_1 and P_2 is given by:

$$\frac{X - X_1}{X_2 - X_1} = \frac{Y - Y_1}{Y_2 - Y_1} = \frac{Z - Z_1}{Z_2 - Z_1}$$

Setting each of these ratios equal to a parameter τ yields the parametric form:

$$X = X_1 + \tau(X_2 - X_1)$$

$$Y = Y_1 + \tau(Y_2 - Y_1)$$

$$Z = Z_1 + \tau(Z_2 - Z_1)$$

P_1 and P_2 are so chosen that P_1 is the light source position and thus the origin of the system. This reduces the equations to:

$$X = \tau X_2$$

$$Y = \tau Y_2$$

$$Z = \tau Z_2$$

The parameter τ has the following useful properties:

$\tau > 1 \Rightarrow$ P is on the extension of $\overline{P_1P_2}$

$\tau = 1 \Rightarrow$ P is identical to P_2

$\tau = 0 \Rightarrow$ P is identical to P_1

$0 < \tau < 1 \Rightarrow$ P is between P_1 and P_2

$\tau < 0 \Rightarrow$ P is on the extension of P_2P_1

In addition to providing the coordinates of projected points, τ can be used in establishing the truth value of the relations I s J and I $\rightarrow$ J. As we see in Figure 5, it is necessary to use the values of τ for each vertex to see if the shadow makes sense. In this case polygon π_i projects onto the plane of polygon π_j as P_1', P_2', P_3', P_4'. The routine has to check for such situations and instead use P_1', P_2', P_3'', P_4''. We do not, however, make any checks to see whether the shadow as cast is visible. The fact that two or more polygons may cast shadows that overlap on a given polygon has no effect on the computation. The final stage of the process takes care of such contingencies.

We feel that in further implementations it would be useful to defer actual computation of "shadow key squares" until a complete scanline is processed. In this manner, it would be possible to introduce and compute shadows cast on a polygon only in the case that the polygon was, in fact, visible at some point on the scanline. This technique would eliminate a large number of "shadow key squares" that are, in fact, not needed at all in the production of the picture. Another extension being considered is to allow polygons to have a degree of translucency. However, self-shadowing polygons then would have shadows visible on them and this would cause a large increase in the number of shadow lines.

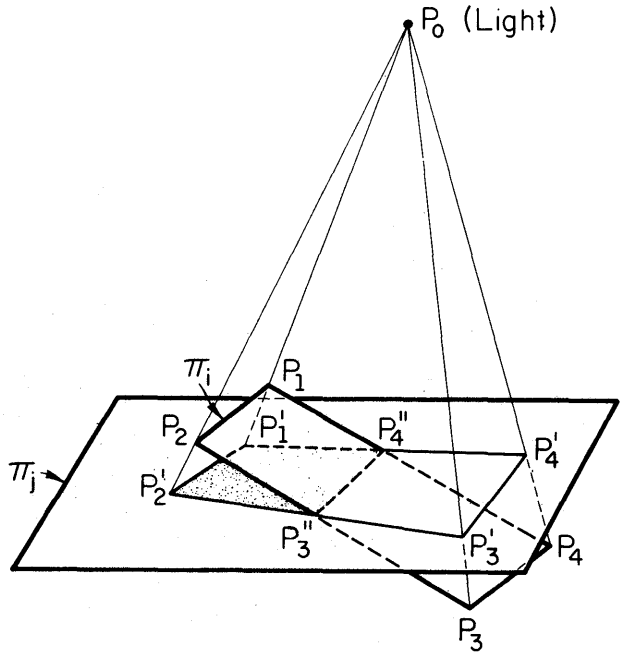


Figure 5—Shadow projection

THE FINAL OUTPUT PROCESS

The final data set for the half-tone image consists of two parts. The first part contains the three-space plane equation coefficients for the surfaces of the three-space structure and the position data for the illumination source. The second part contains the linear string of scan control data items: "key squares", "shadow key squares" and "self-shadow key squares". It is the function of the PIXSCANR routine to assimilate these two masses of data and to produce the final half-tone image.

In order to couple our results closely to the equipment that was available for our use, we modified the display hardware to provide a special raster scanning operation in which the equipment automatically performs the function of stepping across the raster, and our data input specifies what intensity levels will be used in various sections of the scan. The raster is plotted from left-to-right and bottom-to-top on the display screen. A data item initializes the scan to the starting position and gives the initial intensity value for the CRT beam. In addition, the stepping increment δ is given. When the scanning operation comes to the end of a scanning line, the value of the x coordinate is reset to 0 and the y coordinate is incremented by δ.

The remaining data items presented to the display hardware consist of an x coordinate value and an intensity value. As the scanning operation proceeds, the current x coordinate value of the scan is compared to the x coordinate of the next data item. If agreement is achieved, then the intensity of the beam is adjusted to the new value and the scan continues. Once set, the beam intensity does not change.

This raster scan operation allows the final image to be exposed with a minimum number of actual data items being sent to the display hardware. A moderately complex picture might have, for example, an average of 20 intensity changes per scanline. If it were necessary to send display information for every point in the picture, each line would have 512, 1024, 2048, ... or more data items associated with each scanline. Another benefit gained from this condensed data format is that the amount of data needed per picture varies in a

linear manner with the size of raster being used. Computation speed also varies in a linear manner.

The section of PIXSCANR which processes the first part of the data file from the shadow-tone algorithm establishes the intensity functions associated with the polygons of the three-space structure. A basic assumption made in our current implementation is that the intensity of light reflected from a given planar surface is uniform over the entire surface. Although this does not hold true in the physical world, it is close enough for our purposes.

We selected a cosine function for the intensity of the reflected light from a given surface. A ray emanating from the center of the illumination source and passing through the "centroid" of the surface defines the angle of incidence of the light for the entire plane. The "centroid" is calculated by finding the average value of the x and y coordinates of the vertices of the polygon and solving for the corresponding z using the equation of the associated plane. The cosine of the angle between the surface of the polygon and the ray drawn to the illumination source is then given by:

$$\cos\theta = \frac{|\,Aa + Bb + Cc\,|}{\sqrt{A^2 + B^2 + C^2}\sqrt{a^2 + b^2 + c^2}}$$

where the A, B, C are coefficients of the plane equation and a, b, c, are direction numbers of the ray. The intensity of the reflected light from the surface of a polygon not in shadow is given by:

$$I_i = |\cos\theta| * R_i * \text{RANGE} + \text{IMIN}$$

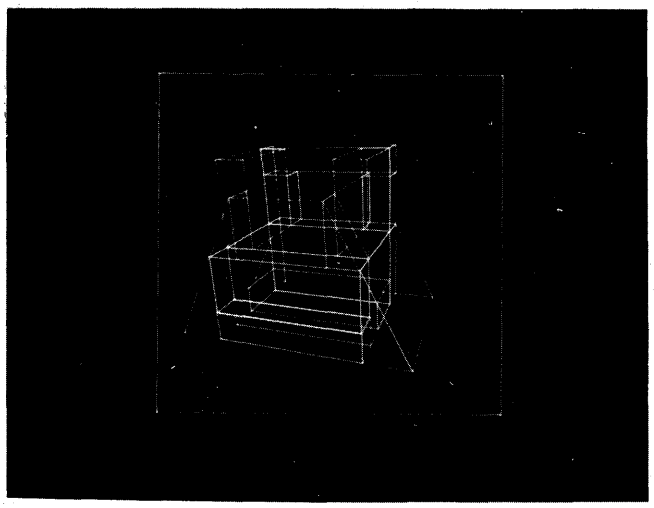
RANGE and IMIN are parameters controlled by the user which specify the total range of intensity to be used in the half-tone image and a translation of that range along the scale of the display hardware. R_i is a pseudo-reflectivity coefficient specified by the user for each polygon surface to allow some differentiation between surfaces. Those polygons indicated by "self-shadow key squares" are assigned a special intensity due to "ambient" light. This intensity is given by:

$$I_{ss} = 0.2\,\frac{C}{\sqrt{A^2 + B^2 + C^2}}\,\text{RANGE} + \text{IMIN}$$

where A, B, C are coefficients of the equation of the plane of the self-shadowed polygon.

Once the intensity functions have been calculated, the processing of the "key squares" data set begins. In the original version of PIXSCANR used for non-shadow half-tone image presentations, it was a simple matter to transpose the "key squares" directly into data items to send to the display hardware. For the shadow half-tone system, the addition of the "shadow key squares" and the "self-shadow key squares" to the data set complicates the process immensely. Recall that the function of keeping track of which shadows are being pierced by the scanning ray on a scanline is now a proper operation to be performed by PIXSCANR.

Shadow tracking is accomplished in an $n \times n$ binary array, in which the (i, j)th position is a 1 if polygon j is casting a shadow on polygon i. As the "shadow key squares" are processed from the data file, the associated positions in the binary array are flipped from the in-shadow state to the out-of-shadow state. When a structure "key square" is processed, the intensity of the



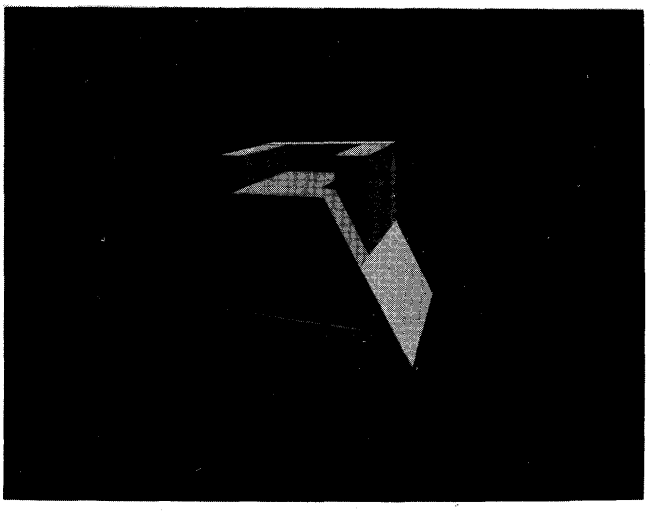


Figure 6—Two presentations of a three-space structure

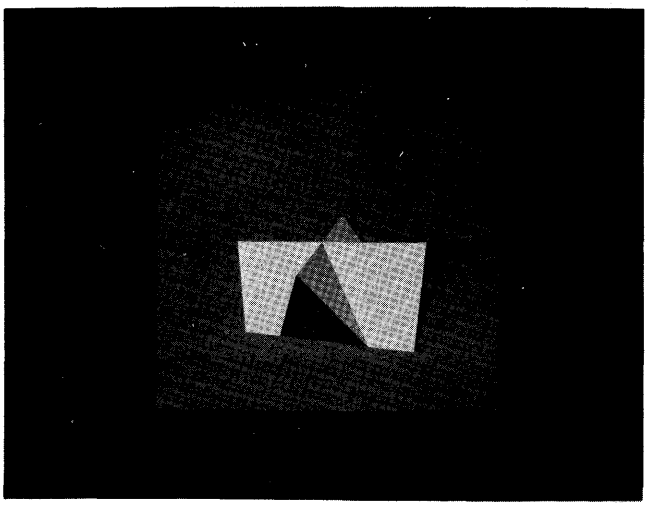


Figure 7a—A-Frame cottage with no shadows

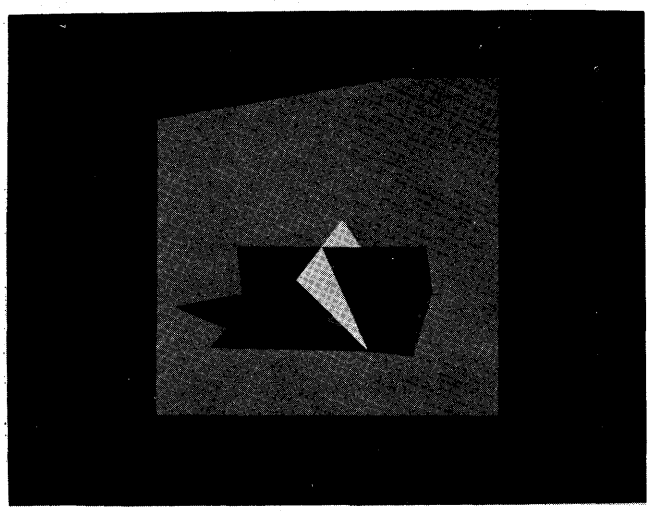


Figure 7b—A-Frame cottage with shadows

beam will be set at that point. Otherwise, the shadow will be indicated by using the minimum value of intensity for the image (IMIN).

The remainder of the operations performed by the PIXSCANR routine are concerned with the outputting of the final image on various photographic media. At CSL, we have the option of photographic recording on either Polaroid 3000 speed black and white film or 70mm RAR type 2479 recording film. The PIXSCANR routine also provides for inversion of the image in a complementing operation on the intensity functions. Further output capability is provided for making animation sequences on a 16mm animation camera.

RESULTS OF THE ALGORITHM

The two photographs of Figure 6 compare the "wire-frame version" of a three-space structure with the shadow half-tone presentation of the same structure. The object consists of three parts, all arranged to fit interlockingly within one another. The computation time for our implementation on the CDC 1604 computer was about 2 minutes, 20 seconds. Time for computation of the non-shadow half-tone presentation ran about 45 seconds.

Figure 7 shows the same view of an A-frame summer cottage, first with no cast shadows in part a. and then with cast shadows in part b. Non-shadow half-tone computation took 13.5 seconds and the shadow half-tone computation required about 27.0 seconds. Both cases indicate that time required for shadow half-tone computations is about twice the time required for the non-shadow case.

As an example of how the same scene appears with the light source in various locations, Figure 8 shows the A-frame cottage in three different appearances. When only the light source position changes, only the shadow

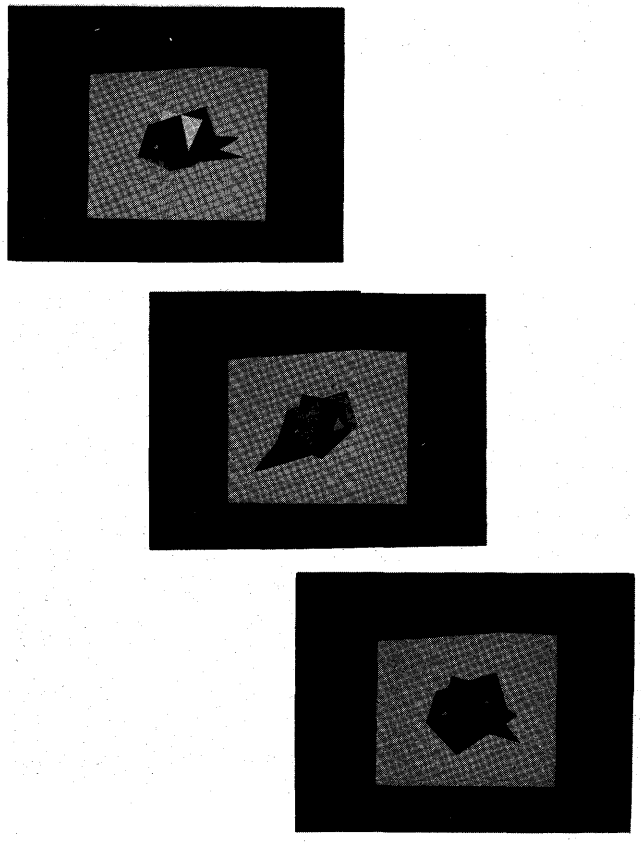
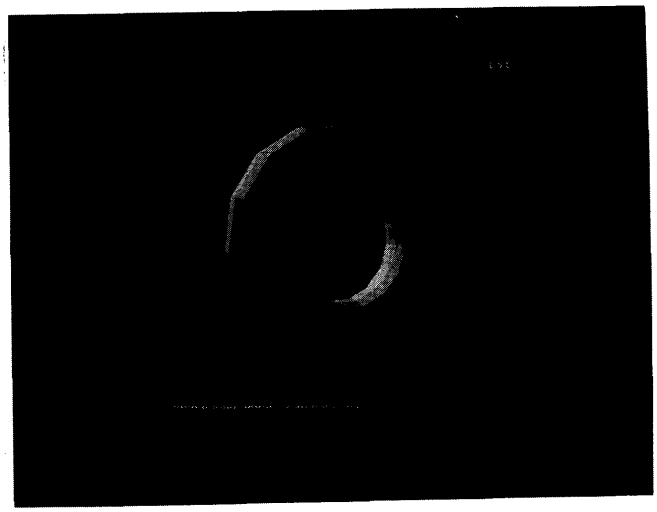


Figure 8—A-Frame with different light source positions

pairings and their subsequent computations change. Thus, future implementation of shadow half-tone algorithms may be able to save computation time by passing the "key square" data from scene to scene and computing only the "shadow key squares" and "self-shadow key squares".

If the only change made from one presentation to another is a movement of the observer position, the converse of the above occurs. The shadow data does not change and only the "key square" data must be computed. Both of these attempts to reduce computation time by borrowing from past results will require increased amounts of storage and techniques for merging the old and new data sets to generate the final half-tone image.



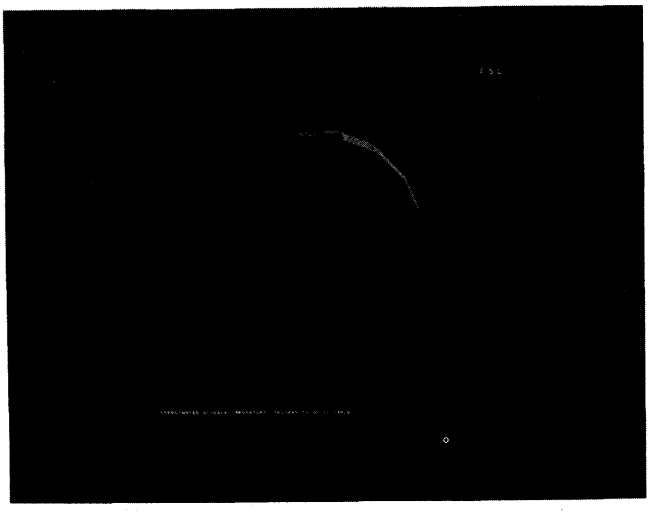


Figure 9—Back-lighted Torus

Our final presentation of Figure 9 shows a torus in free space back-lighted with respect to the observer position. The torus is constructed of 225 planar polygons. The time for computation of the non-shadow case was one minute, 25 seconds. The shadow half-tone image required about three minutes of computation. Approximately 700 shadow pairings were found to be useful by the detection stage of the algorithm. Only a small number were actually detected in the final image. In fact, only by back-lighting the torus could the complete image be processed because the number of visible shadow exceeded the limits of storage available during execution of the program. Efficient computation of the final image data will depend upon the availability of sufficient amounts of direct address core storage or an auxilliary storage medium which can be accessed in speeds approaching main memory access time.

BIBLIOGRAPHY

1 C WYLIE G ROMNEY D EVANS A ERDAHL
Half-tone perspective drawings by computer
Proc of the Fall Joint Computer Conference Vol 31 49–58 1967

2 J WARNOCK
A hidden line algorithm for half-tone picture presentation
Tech Report 4–5 University of Utah Salt Lake City Utah May 1969

3 B ELSON
Color TV generated by computer to evaluate spaceborne systems
Aviation Week and Space Technology October 1967

4 J BOUKNIGHT
An improved procedure for generation of half-tone computer graphics presentations
Report R-432 Coordinated Science Laboratory University of Illinois Urbana Illinois September 1969

5 A APPEL
Some techniques for shading machine renderings of solids
Proc of the Spring Joint Computer Conference Vol 32 p 37–49 1968

6 A APPEL
The notion of quantitative invisibility and the machine rendering of solids
Proc ACM Vol 14 p 387–393 1967

7 M KNOWLES
A shadow algorithm for computer graphics
Department of Computer Science File No 811 University of Illinois Urbana Illinois 1969

The case for a generalized graphic problem solver*

by E. H. SIBLEY, R. W. TAYLOR, and W. L. ASH

University of Michigan
Ann Arbor, Michigan

INTRODUCTION

Not so many years ago, *SKETCHPAD*[1] and DAC[2] set the whole world of computing on a philosophical bender. People, who either knew little of the subject or else should have known better, started talking up a storm. The whole of engineering was about to be revolutionized and everyone should prepare *now* or be sunk, to drown in their own ignorance.

Unfortunately, even though we, in computation, have been regularly beset by super-salesmen who keep on telling us "how good its going to be," we still are suckers for a good line. We sat at the edge of our chairs listening to the prophets, and later scurried around trying to learn more about the wonders of the future, and bought expensive hardware (which we didn't yet know how to use) so that we should be ready.

Fortunately, some business managers finally asked why the expensive equipment was sitting there, and as a result, many people moved away from "research-like" operations to a more reasonable "application" approach. This meant that the people in graphics became divided into two camps (almost mutually exclusive) who either tried rather unsuccessfully to implement a generalized graphic system, or else tried to produce a working application program. The latter set of investigators have produced many useful packages, or we should all have watched the graphics hardware being converted into TV sets. Why then has the well promoted "generalized graphic system package" proved so elusive? In this article, we shall try to show that this is due to lack of knowledge on the part of the prophets. That in fact they were proposing systems which needed, as their *core* a *generalized problem solver*, which is, after all, only asking for a really good artificial intelligence package, and that field has been having its own problems for years, too.

The rest of this article focuses on what *has* been done, and what reasonably *could* soon be done towards providing a useful package for generalized graphic problem solving. The techniques adopted in SKETCHPAD's constraint satisfactions are discussed with conclusions made about their generalization and their wider application in other fields.

Finally, some of those scientific, engineering, and mathematical modeling techniques which normally use graphics for either visualization, analysis, or numerical computation of the solution are examined in some detail. The conclusions suggest that there is still something that can be achieved in the near future, although it is probably less than many of the early optimists had predicted.

BACKGROUND AND JUSTIFICATION

Before starting a discussion on the merits or deficiencies of a *generalized graphic problem solver*, we must define that term. The use of graphics (other than symbols) for the solution of problems is common throughout much of engineering and science, and even in some fields of mathematics. The concept is often associated with the idea of a model which involves a topological or physically scaled picture from which the original problem is solved. Sometimes the picture itself is the solution (e.g., in the case of computer generated art, some phases of architecture, etc.), but more often the picture is either an immediate model from which algebraic or numeric equations are produced (these are solved to give the answer) or else the picture is later used to generate information (e.g., for architecture, the original drawings may later be examined to give material and cost information). Now if we consider a software computer system which aids the engineer, scientist, or mathematician in the formulation and

*The work reported in this paper was supported in part by the Concomp Project, a University of Michigan Research Program Sponsored by the Advanced Research Projects Agency, under Contract Number DA-49-083 OSA-3050.

analysis of a range of problems, using the heuristics of the man to formulate a reasonable model, and the computer to aid in and augment the analysis, etc., then this software could be termed a "generalized graphic problem solver".

At the other end of the scale from the generalized graphics processor, is the specialized graphic package. Here the classical input–output devices (e.g., card reader and printer) are replaced by graphic devices (e.g., light pen and CRT) so that the user has an easier time stating his problem or understanding his solution, probably because it is two dimensional in form, or nearer to the engineers' medium, viz, pen and paper.

The first question must then be: "Is generalized graphics desirable?"

Obviously all the early graphic-systems/computer-aided design prophets thought that it was. To quote one:[3]

"In the near future—perhaps within five and surely within 10 years—a handful of engineer-designers will be able to sit at individual consoles connected to a large computer complex. They will have the full power of the computer at their fingertips and will be able to perform with ease the innovative functions of design and the mathematical processes of analysis, and they will even be able to effect through the computer, the manufacture of the product of their efforts. . . ."

Recently, there has been some retrenchment:[4]

"Suddenly a new world seemed to have sprung into being, in which engineers and architects could sit in front of a screen . . . and conjure up automobiles or hospitals complete in every detail, in the course of an afternoon. Unfortunately, reality turned out to be more elusive than some people expected. . . . What emerges from the above is a requirement for a general system for building models, to which can be applied transformations and algorithmic procedures. . . ."

On the whole though, there is still much to be said for generalized graphics systems research. To begin, we can state the rather overoptimistic argument that: there is bound to be some practical or more efficient fallout from research in general systems; thus we will have better specialized systems even if we never get a really generalized one. In this argument, we are probably on reasonable ground, since the special systems of today have nearly all been spin-offs from the past generalized systems. But this is not enough.

The important point seems to be that a reasonable research effort can and will produce further steps towards a generalized system, or at least a "less specialized" system.

A second question is:

If we had a generalized graphic system today, could industry afford to use it? Unfortunately, with our best hopes, we still have to admit that the answer is NO. This is mainly due to the fact that specialized systems are normally cheaper, and often easier to run than a generalized system. This could be compared to the difference between a "FORTRAN machine" which provides a useful but *ad hoc* language, and a Turing machine, which is certainly general, but is by no means as easy to use.

How, then, do we hope to succeed? The first way is by developing more powerful techniques, which, though general, are easy to use and well interfaced to the user. The second is the normal effect of engineering progress and the economies of scale. As time goes on, we might hope to see both cheaper hardware and a ready pool of useful routines from a user community which can be integrated into a total, but generalized system.

THE SKETCHPAD METHOD OF CONSTRAINT SATISFACTION

Since picture meaning and solution will form the heart of any generalized system, we felt that a deeper understanding of the "graphical constraint problem" was necessary. Naturally, we started with the SKETCHPAD approach. We had the following questions in mind:

1. Were SKETCHPAD's methods as general as some claimed or were people misconstruing some exciting beginnings?
2. Why hadn't further extensions of the graphical constraint problem appeared?
3. How might we extend the SKETCHPAD work on graphical constraints?

Some answers which led to our general conclusion will appear in the next few sections. It will be helpful to consider the SKETCHPAD method in some detail for background purposes.

When SKETCHPAD was commanded to satisfy a set of constraints on a picture, it did so by using numerical measures of how much certain "variables" (usually points) were "out of line" (our phrase). Constraint satisfaction was therefore a matter of reducing the various numerical error measures to zero. The errors were computed by calling, for each constraint type and for each degree of freedom restricted by that constraint type, an error computing subroutine. Each subroutine would compute an error "nearly proportional to the distance by which a variable was removed from its proper position". Thus if the components of a

variable were displaced slightly and the error subroutine called for each displacement, a set of linear equations could be found. These equations had the form

$$\sum_i \frac{E}{x_i}(x_i - x_{io}) = -E_o$$

where x_i is a component of a variable, E is the computed error, and the subscript o denotes an initial value. This set of equations could then be solved by Least-Mean-Squared Error Techniques to yield a new value for each component involved.

The general constraint procedure was thus based on this numerical measure, and the most general algorithm available was relaxation. Thus a variable was chosen and re-evaluated using the LMS technique such that the total error introduced by all constraints in the system was reduced. The process continued iteratively and eventually terminated when the total computed error became minimal.

Obviously, the relaxation method, with a set of equations to be computed and solved many times, was slow. Thus, before using relaxation, SKETCHPAD employed a heuristic, which will be discussed in some detail because we believe many similar techniques will be necessary. The object of the heuristic was to find an order in which a set of variables could be re-evaluated such that no subsequent re-evaluation would affect a former one. If this order could be found, then the constraint satisfaction process could proceed much faster because the variables involved would need only one re-evaluation.

To perform this search, the user picked a starting variable. SKETCHPAD then considered the constraints on that variable and formed the set of all variables which participated with the starting variable in the constraints involved. If some of these variables had sufficient degrees of freedom to satisfy the constraints, then one could be sure that they would not affect previous constraint re-evaluations. Thus such constraints could be removed from the constraint set on the starting variable, possibly allowing it to be easily re-evaluated. The technique was extended to build chains of constrained variables until sufficient free ones were found. Of course, such a set of free variables did not always exist, and it was necessary to have the relaxation method as a back-up.

COMMENTS ABOUT SUTHERLAND'S METHOD

In the conclusion to Reference 1, Sutherland stated, "It is worthwhile to make drawings on a computer only if you get more out of the drawing than just a drawing" (p. 67). While this statement might arouse some debate (are plotters useless?), it nevertheless reflects the key point that one must be able to associate meaning, and thus constraints, with a picture. When the Sketchpad method is evaluated relative to arbitrary picture semantics, several observations can be made.

First, all constraints in SKETCHPAD were eventually related to a numerical error definition having to do with distance. This technique was clearly oriented toward the LMS equation solution method, but in addition had several advantages. It allowed new constraint types to be added quickly since all one had to do was write a new set of error computing subroutines; the solution machinery was already present in the overall system. The solution technique itself, though it involved a preliminary heuristic, could almost guarantee a solution in reasonable if not entirely pleasing time. In addition, the numerical approach together with relaxation yielded results for a class of problems that were of particular interest in the author's field. The approach was therefore eminently successful *for certain operations.*

In a more general context, the approach has some drawbacks. The relaxation method depends critically on the results of a previous iteration. Thus once having applied it, it is a non-trivial matter to restore the picture to its previous status; one might have to make a copy of parts of storage, for instance. Design by trial and error, with recovery from undesirable conditions, is thus hampered. Furthermore, the relaxation method is not generally selective in picking some priority in variable re-evaluation. One constraint is as strong as another and all are broken with equal abandon. This runs counter to most realistic concepts of a constraint. These drawbacks were known to the author: "There is much room for improvement in the relaxation process and in making 'intelligent' generalizations that permit humans to capitalize on symmetry and eliminate redundancy" (p. 55).

A final criticism of the numerical definition of constraints centers on the degree to which they are "picture oriented." A review of the available constraint types in SKETCHPAD (Appendix A) quickly reveals their obvious correlation with "picture transformations". Certainly such picture transformations are a desirable part of man-machine communication. However, if the system is limited to constraint satisfaction methods involving such simple picture transformations, the system capabilities are undoubtedly themselves limited. Examples will be presented later. The point is that constraints are a function of picture semantics, and only incidentally of picture geometries.

It should be noted that the above criticisms are

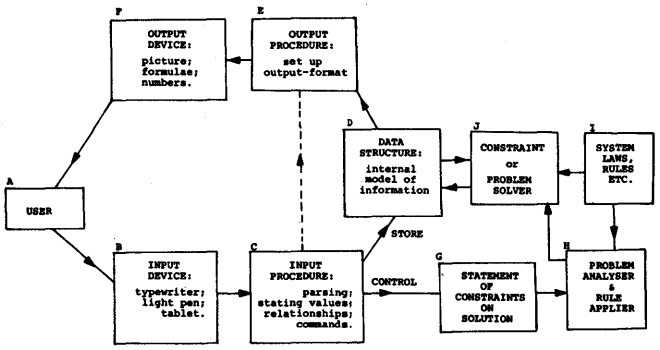


Figure 1—Generalized (graphic) problem solver

centered on the mapping from constraints to numbers. The philosophy of relaxation and especially the preliminary heuristic do not depend critically on the numerical error computing subroutines. They are, rather, general strategies for solution. Relaxation could be rephrased as:

"Transform object A into object B"
"Reduce the difference D between object A and object B"
"Apply operation Q to object A."

which the reader may recognize as a strategy in the Newell and Simon GPS program.[7] In such a context, the preliminary heuristic becomes a method for ordering how the transformations will be effected. GPS has similar, though less dynamic, ordering strategies.

Thus while the Sketchpad approach was an exciting and useful beginning in the area of graphical constraints, it should not be taken as the last word. The author himself has stated, "Much room is left in Sketchpad for improvements. . . . A method should be devised for defining and applying changes which involve removing some parts of the object drawing as well as adding new ones" (p. 70).

The last sentence of the quotation suggests the generality discussed above. It implies more general picture operations, macroscopic in nature; it is a short step to view these macros as a "subproblem" strategy.

THE GENERALIZED GRAPHIC PROBLEM SOLVER

In analyzing the difference between special graphic systems and generalized graphic problem solvers, we found that the information flow chart for both systems, and indeed for many other types of computer-aided design systems, could be represented by Figure 1. Although this figure bears strong family resemblance to work of others,[5,6] it has special new features, and leads to certain unique conclusions.

First let us consider the work of a simple graphic "drawing" program, where the user can merely input a restricted set of graphic entities (such as lines, arcs, etc.), and then view them on a CRT. In essence, he is working with the restricted system A → B → C − D → E → F → A; he is unaware of the data structure representing his picture, and is unable to either affect it to produce a new drawing, or analyze its meaning (except, possibly, as an artistic entity). The particular blocks that he uses are:

B: An input device such as a light pen, or tablet, or even a string of characters from a teletype which represent predefined graphical items.
C: An input procedure, not necessarily highly sophisticated, nor particularly general in its parsing rules, which stores the information.
D: A data structure, normally unsophisticated, which represents the input information in a somewhat compacted form, possibly in hierarchic classes called pictures.
E: An output procedure, which can take information about a given picture from the data structure, and format it so that the output device will be able to use it (e.g., place it, in correct format, into a "display file").
F: An output device, probably a digital or analog driven cathode ray tube.

If we now consider what additional information flows in SKETCHPAD, we see that the loop ABCDEFA is still needed, with much greater sophistication in blocks C and D, but also that other parts must exist. Besides the drawing function, we have the *constraint* function; e.g., two lines are stated to be parallel. The additional functional requirements are therefore:

G: A procedure which interprets (or parses) the constraining command
H: A process which (possibly heuristically) determines the particular set of rules which are to apply for this additional constraint
I: The set of rules that can apply for the particular branch of science or mathematics being modeled. In this case, only geometry.
J: A procedure which takes the new rule, and, in conjunction with all previous rules laid down by the user, produces the final result, thereby potentially changing the data structure (D).

As an example, suppose we enter two lines into SKETCHPAD. This will cause two loops through ABCDEFA. We now state that these must be parallel.

The arc ABCGH determines that there is a given rule in I to satisfy the condition "parallel"; i.e., that the slopes of the lines be the same. J now determines the error metric, and minimizes this, finally producing new definitions for the two lines in D. The new result to the user (A) is given via E, F.

Ross[5,6] working from a similar system conception, argued that it would be pointless to write a family of special purpose packages, since these could never satisfy a general user population. Nevertheless, he realized that boxes H, I, and J were the key to any useful system. He therefore proposed (and has since built) a system for building systems. The idea is that a sophisticated user, provided with general packages for building boxes B, C, D, E, and F, and further provided with a general language for building boxes H, I, and J, could produce specialized packages economically. Our conclusions differ from Ross', but before examining them, we should draw parallels to other (non-graphic) systems.

For a numeric problem, e.g., the solution of supersonic flow in a divergent nozzle, we have no automatic display loop. The information flow is essentially ABC, (D and G), (H and I), J, D, E, F, though some of the steps are either eliminated or merged with one another; e.g., G, H, and I are merely reformatting the input so that it can be worked on by a special purpose program J (which is a differential equation solver and special purpose output formatter).

For algebraic analysis, we have a surprisingly similar set of procedures. The input (BC) and output (EF) programs are oriented towards equations, and the parsing in the input procedures (from C to D) may be quite complex, but the constraints on the solution (G) are now the selection of a subset of the equations, probably by user command, with some criteria (such as elimination of a variable) for the solution. The analyzer and rule applier (H) now must select, from the set of system laws (I) or axioms or rules, that set of rules and their order of application so that a reasonable solution can be generated when they are used by the problem solver (J) to produce the required set of operations to provide a solution.

Finally, we are ready to make our generalization, which is that graphic problem solvers, like general problem solvers, are in need of good heuristic solution techniques. Thus the conclusions are:

(i) The constraint satisfaction problem, though capable of solution for simple graphic pictures with a few constraints by algorithmic or simple maze solving techniques, is in fact a specialization (though a not much easier problem) of the general problem solver.

(ii) That if a graphic system were built with the GHJ of Figure 1 replaced by a General Problem Solver, then the "constraint problem" would only be one of many useful operations available, and that it would then be possible to use this GPS, examining the data structure and a set of rules (possibly for topology and model building in electronic circuits) to analyze the physics of the model, either numerically or symbolically.

(iii) It then would be possible for the user to be truly in charge of his model. In contradistinction to other writers,[5,6] we believe the correct interface between the user and his modeling is at "B" as a user of a given system, or at "I" when he is altering his system rules or setting up his system for a new type of model (e.g., mechanical engineering instead of electrical, etc.).

Examples of use of a generalized graphic problem solver

Let us consider the operation of such a generalized (graphic) problem solver on two problems from engineering (Figure 2). The first of these represents an electric circuit problem solved using a "phasor diagram". Naturally, this is a problem which is "easily" solved using a special purpose program. First, we construct the circuit diagram by using a series of lines to connect resistors and capacitors which have been pro-

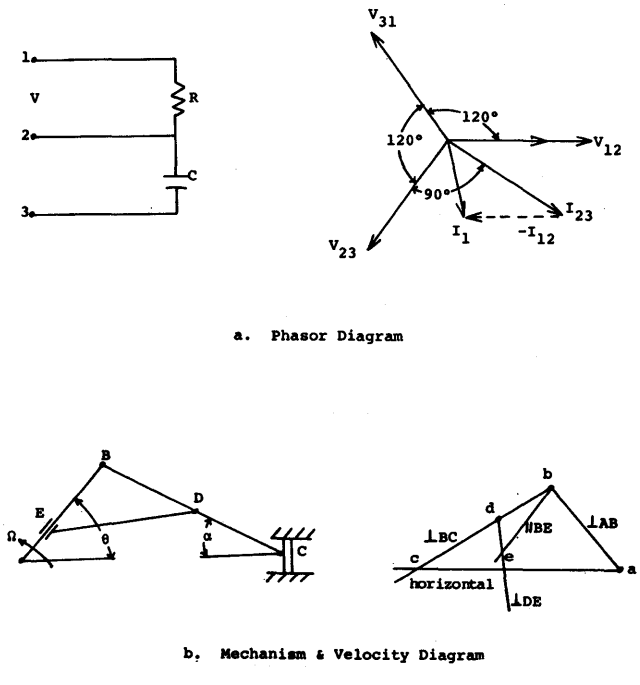


Figure 2—Examples for a generalized (graphic) problem solver

vided as "subroutines" by previous users or system designers. We associate with the resistor the "numerical" value R, and with the capacitor the value C. If the idea of "three phase balanced voltage" has been previously introduced, we associate 1, 2, 3 with the lines and give V as the line voltage. All of the drawing and associations have been entered either by light pen or typewriter key strokes. The picture has been drawn using loop ABCDEF, and all associations are passive constraints which have not caused any "problem solution" up to this point.

It is, of course, possible that the engineer, wishing to be "neat" positioned the lines horizontally or vertically, but this was either a constraint *during* drawing, and hence automatically satisfied, or else the constraint was later applied, and the GPS loops of (C, D), G, (H, I), J, D will have no real difficulty in resolving the problem since the problem is not overconstrained, and any one constraint can be selected to be applied first with no effect on the outcome.

If "phasor diagrams" and the laws of impedance are already a part of the electrical-bag-of-tricks, then the diagram to the right could immediately be constructed, with the value of I_{12} set as (V/R) in phase with V_{12}, and the value of I_{23} set at (VwC) leading V_{23} by 90° (where w is the angular velocity of the applied voltage). All that remains is the need for Kirchhoff's law to produce I_2 as the vector sum of I_{23} and $(-I_{12})$ . . . this also requires the concept of a triangle for solution.

If the previous stored experience (the system laws) does not include phasor diagrams, the user may still produce a solution by applying the graphic construction phase as follows: Draw V_{12} *horizontally*, define its length to be V (non numeric scales being allowed), and the other voltages V_{23} and V_{31} at 120° and 240° respectively anticlockwise. In the same way, I_{12} would be defined parallel to V_{12} (with scaling) and I_{23} perpendicular to V_{23}, clockwise.

If the values of V, R, C, w are known, then actual numeric values of the current (and phase angle) can be computed. If the user wishes, he could use an algebraic solver to obtain I_{12} as $(V/R) \angle 0°$, I_{23} as $(VwC) \angle -30°$, and even I_2 as $V\{(\sqrt{3}/2)wC - R^{-1} - i(wC/2)\}$ where i is the 90° operator (assuming that this notation was available).

As an extension of this type of program, we can introduce topological considerations to include the concept of *parallel* and *series* with their associated algebraic or numerical transforms. It would also be possible to apply more complicated transformation rules, such as $T \rightarrow \pi$ and its inverse. Obviously, the simple rules of complex number manipulation should be included in this "circuit package".

The second example of Figure 2 involves the position and velocity of a simple four-link mechanism. First, we draw the mechanism, constraining each length fixed, point C as moving horizontally, point E as moving on AB, and D as fixed between B and D. The generalized graphic problem solver could produce a "picture" of the mechanism for any given angle of AB from the horizontal. As a result, we could plot if we wish the curve of displacement-v-angle. Alternatively, if we have an algebraic/trigonometric processor available, we could produce equations for the positions of the various points of the mechanism (e.g., $B = l\cos\theta + il\sin\theta$, etc.). This would probably also introduce the angles of BC and DE from the horizontal, with "constraint" equations for determining these as functions of the various lengths and θ (e.g., $BC/\sin\theta = BA/\sin\alpha$).

One common method of solving for the velocity is to draw a diagram, as shown with lengths proportional to the velocity, and the relationships of the angles either along or perpendicular to the special diagram. Thus ab = ΩAB to scale, and perpendicular to AB. To find C, we intersect bc (drawn ⊥ BC) and ac (drawn along the direction of constrained piston motion, i.e., horizontally). The position of d is determined by preserving ratios, i.e., bd/bc = BD/BC. Thus the velocity diagram can be constructed, and the velocity of all parts found.

Obviously, another way to solve this problem, given the trigonometric rules, laws, etc., above, and also a simple set of differentiation rules, would be to solve for the time derivative of the displacements, and hence obtain a closed form solution.

CONCLUSIONS

We have seen, in the two examples, that there are several ways that the same problem can be tackled, given a generalized graphic problem solver. But indeed, this should not be surprising, because we have already assumed that a significant feature of the generalized *graphic* problem solver is a general problem solver. We also know that much of engineering graphics in the past was aimed at either visualizing the model in such a way that it could be solved, or else providing a way for obtaining numerical solutions by scaled drawings.

This may now lead us to two important facts. The first is: scientists and engineers have, in the past, made significant use of graphic diagrams to solve mechanical, electrical and other engineering and scientific problems. Is it reasonable to use these techniques, when they represent only a *man* oriented solution, and when a machine might make the solution more simply using the algebraic formulation? In some ways, this question

is academic, since any *algorithm*, which is easily defined, should be welcome to the computer user. In fact, these sorts of techniques are highly man-machine or symbiotically oriented. Strangely enough, the algebraic approach would be much less man controlled, needing heuristics to determine how to solve the series of equations (i.e., to determine the order of applying the laws or rules). Thus we see, once again, that the future betokens more applied artificial intelligence.

The second is:

Although many investigators have looked on the world as man-machine oriented, Figure 1 suggests that this cannot be the case if we plan to expand towards generalized systems. This is because the user only appears in one loop (ABCDEFA), and is excluded from the other (GHJD). When the problem is being solved, the user *may* be able to assist, but more often, the machine representation of the problem and its present state of solution may be unintelligible or untranslatable to the man. This does not mean that there are *no* places where the man could help, but it suggests that there is no single successful technique where a man can help. An illustration in Reference 8 shows that if graphic procedures are called from within other procedures, then the human decision maker could be confused by a "question" or call-for-aid generated in a low-level subroutine, since the user may not even be aware of the conflict, let alone what it means.

Finally, to end on an optimistic note:

The graphic systems which are specialized are significantly different from the generalized systems, but they are, nonetheless similar in many parts to the generalized system described here. Many of the routine manipulations in a special system are still needed in the more general. The total general system involves generalized problem solvers which, though being developed in several locations in the country, are still very primitive; however, recent work on both the theoretical and practical level[9,10] suggests their ultimate utility. But should all else fail for the next few years, we can still fall back on the semi-automatic procedures of the engineer/scientist, like those discussed in the last section and shown in Figure 2. Although this is not a large step forward, we have plenty of room for research even in this cut-down version, while we are waiting for AI to develop.

REFERENCES

1 I E SUTHERLAND
SKETCHPAD: A man-machine graphical communication system
Lincoln Laboratory Technical Report 296 Lexington Massachusetts January 1963

2 E L JACKS
A laboratory for the study of graphical man-machine communications
Proc of the Fall Joint Computer Conference Vol 26 Part I p 343 1964

3 S A COONS
Computers in technology
In Information W H Freeman and Company San Francisco California 1966

4 J C GRAY
Compound data structure for computer-aided design: A survey
Proc 22nd National Conference ACM p 355 1967

5 D T ROSS
The AED approach to generalized computer-aided design
Proc 22nd National Conference ACM p 367 1967

6 D T ROSS J E RODRIQUEZ
Theoretical foundations for a computer-aided design system
Proc of the Spring Joint Computer Conference p 305–322 1963

7 A NEWELL H A SIMON
GPS, a program that simulation human thought
In Feigenbaum and Feldman Computers and Thought p 279–293

8 E H SIBLEY
The use of a graphic language to generate graphic procedures
Proceedings of Second Illinois Conference on Pertinent Concepts in Computer Graphics April 1969

9 G W ERNST
Sufficient conditions for the success of GPS
JACM Vol 16 No 4 October 1969

10 C C GREEN
Application of theorem proving to problem solving
IJCAI Washington May 1969

A variance reduction technique for hybrid computer generated random walk solutions of partial differential equations

by DR. EVERETT L. JOHNSON

The Boeing Company
Wichita, Kansas

INTRODUCTION

The work done to date on the analog/hybrid Monte Carlo solutions of partial differential equations can be summarized by reviewing three works: one research report and two Ph.D. theses.

Chuang, Kazda, and Windenecht were the first to demonstrate the feasibility of a Monte Carlo solution of a class of partial differential equations on an analog computer.[1] The boundary value problems for which their stochastic solution technique is applicable belong to a family of generalized Dirichlet problems of the form

$$D_1 \frac{\partial^2 f}{\partial x_1^2} + D_2 \frac{\partial^2 f}{\partial x_2^2} - K_1(x_1, x_2) \frac{\partial f}{\partial x_1} - K_2(x_1, x_2) \frac{\partial f}{\partial x_2} = 0 \tag{1}$$

where $K_1(x_1, x_2)$ and $K_2(x_1, x_2)$ are arbitrary functions of x_1 and x_2. The boundary, c, is an arbitrary, finite closed curve—a Jordan curve.

The boundary-value function $\phi(x_1, x_2)$ is a bounded, single-valued piecewise continuous function of x_1 and x_2. D_1 and D_2 are constants.

The method developed is based on the direct relation that exists between partial differential equations and the random process that arises in the analysis of electric circuits subjected to random excitations.[2]

The electrical equations to be simulated for the solution of Equation 1 are

$$\begin{aligned} \frac{dx_1}{dt} + K_1(x_1, x_2) &= N_1(t) \\ \frac{dx_2}{dt} + K_2(x_1, x_2) &= N_2(t) \end{aligned} \tag{2}$$

where $N_1(t)$ and $N_2(t)$ are noise generators with Gaussian amplitude distribution and power spectral densities D_1 and D_2 respectively. Figure 1 demonstrates the simulation technique.

Fundamental to the research of Chuang, et al, is the theorem of Petrowsky[3] which guarantees the convergence of the stochastically obtained solution to that of the generalized Dirichlet problem, Equation 1.

The solutions to several one and two dimensional problems were given in the paper by Chuang, et al. The solutions were obtained as follows: An analog program as shown in Figure 1 was patched. The initial conditions of integrators 1 and 2 were set to the coordinates of the point for which the solution was desired. Boundary crossings were detected by using an oscilloscope, bounded region mask, and photo tube. Upon detecting a boundary crossing, the value of the boundary-value function at the point of intersection was recorded and the process repeated. The approximate solution was then given by

$$f(\xi) = \frac{1}{N} \sum_{n=1}^{N} \phi(S_n) \tag{3}$$

where S_n is the coordinate of the nth crossing, ξ the solution point, and N is the number of repetitions of the procedure. The simulation equipment allowed up to 2000 random walks per hour. Errors were in the 5 percent range and were attributed primarily to statistical variations, presumably in the noise source.

Little[4] extended the class of partial differential equations for which the methods developed by Chuang, et al, are applicable while developing a technique of solution utilizing an analog computer linked to a digital computer. Little solved three types of partial differential equations: parabolic, elliptic, and non-homogeneous.

Little used the analog computer for simulating an electric circuit excited by random noise. The digital

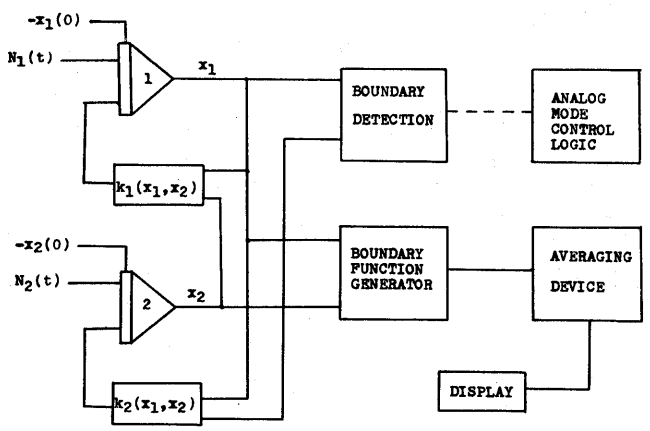


Figure 1—Continuous random walk program

computer collected and averaged the resulting boundary values and controlled the modes of the analog components. With his hybrid system, EAI 231R–V analog computer and Logistics Research Alwac III–E digital computer, he was able to obtain 10 random walks per second. The Monte Carlo solutions compared favorably with the analytical solutions of the example problems.

Handler[5] demonstrated that by use of a high speed repetitive operation analog computer with parallel logic capability, the ASTRAC II, that the Monte Carlo solution techniques developed by Chuang, et al, and Little could be made competitive with more conventional finite difference digital solution techniques. In fact, he demonstrated the ability to plot continuously and directly the solution to partial differential equations. This was accomplished by slowly changing the coordinates of the point for which the solution was desired while performing 1000 random walks per second. The averaging of the intersected boundary values was done by a simple analog averaging circuit.

A. W. Marshall[6] has stated that if a random sampling method is to be used to solve a problem, attention should be turned to three topics.

(1) Choosing or modeling the probability process to be sampled (in some cases this means choice of the analog; in others a choice between alternative probability models of the same process).
(2) Deciding how to generate random variables from some given probability distributions in an efficient way.
(3) Variance reduction techniques, i.e., ways of increasing the efficiency of the estimates obtained from the sampling process.

In summary:

(1) Petrowsky has defined a class of stochastic processes which can be used for obtaining a solution to the Dirichlet problem.
(2) Wang, et al, have established a random process, in the class defined by Petrowsky, to be sampled.

The first two topics suggested by Marshall have been considered previously and satisfactory results obtained. The third topic is the subject of this paper.

The means toward the end will be an examination and implementation of a technique of stratified sampling using the Green's function for a rectangle and for a circle.

The technique is implemented by choosing an approximating region, R_a, which is totally contained within the region, R, for which a solution is desired. Portions of the boundary of R_a may be coincident with the boundary of R. For a point contained in R_a the solution is found by performing walks which originate from the boundary of R_a. The number of walks, N_m, which originate from the mth segment is given by the negative of the normal derivative of the Green's function for R_a integrated over the mth segment. A substantial reduction in variance results from the use of the technique.

A VARIANCE REDUCTION TECHNIQUE FOR THE CONTINUOUS RANDOM WALK

For a solution of Laplace's equation for the region R of Figure 2, the continuous walk technique moves

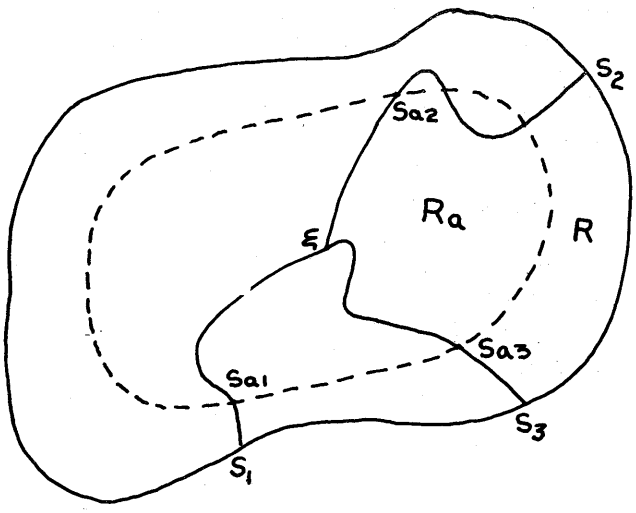


Figure 2—Region of solution with approximating region

continuously from the point of interest, ξ, in the region R until a boundary is intersected. The value of the boundary function $\phi(s)$ is recorded and after N walks the solution is

$$\phi(\xi) = \frac{1}{N} \sum_{i=1}^{N} \phi(s_i). \tag{4}$$

If the Green's function for Laplace's equation and the region R are known then the solution can be written[7] as

$$\phi(\xi) = -\int_c \phi(s) \frac{\partial G}{\partial n} (\xi, s)\, ds \tag{5}$$

where ξ represents the coordinates of the point for which a solution is desired, s is the variable on c the boundary of the region R, and $(\partial G/\partial n)(\xi, s)$ is the derivative of the Green's function with respect to the normal vector of the boundary of R.

In Figure 2 consider the region R_a contained by R with the boundary represented by a dotted line. Each walk leaving the point ξ must intersect the boundary of R_a at least once before intersecting the boundary of R. If a boundary function $\phi_a(s_a)$ were given for R_a, where s_a is the variable on the boundary of R_a, then $\phi_a(s_a)$ evaluated at the points of first intersection with the boundary of R_a allows the solution for Laplace's equation in R_a to be written

$$\phi_a(\xi) = \frac{1}{N} \sum_{i=1}^{N} \phi_a(s_{ai}). \tag{6}$$

It was reasoned that if R_a were a region with a known solution this information might be used to determine the points of intersection on the boundary of R_a for walks originating at ξ. Due to the Markovian nature of the random walk process, walks originating from the boundary of R_a with the correct distribution for continuous walks from ξ should give the solution to Laplace's equation for R. The information sought can be obtained from the normal derivative of the Green's function for Laplace's equation in R_a. The probability density function properties of the Green's function are well established.[8] A heuristic argument is given in Appendix A to support the use of the Green's function to find the proportion of the N walks with origin at ξ which intersect any segment of the boundary of R_a.

If the boundary intersection coordinates are stored during a random walk solution of Laplace's equation for the region R then these values can be used for the solution to Laplace's equation for various boundary functions. The set of coordinate points is then an approximation to the Green's function for the region R and the walk origin ξ. It is important that these intersection coordinates have as nearly as possible the same statistical parameters as the Green's function. The importance is obvious if the boundary function is expressed as a power series.

$$f(s) = a_0 + a_1 s + a_2 s^2 + a_3 s^3 \cdots. \tag{7}$$

The solution to Laplace's equation is then seen to be a function of the moments of s

$$\phi(\xi) = \int_s (a_0 + a_1 s + a_2 s^2 + \cdots) G_n(s)\, ds. \tag{8}$$

The possibility of obtaining a variance reduction in the sample average by originating continuous walks from theoretically determined origins on the boundary of R_a prompted the research reported in this paper.

A STRATIFIED SAMPLING TECHNIQUE USING GREEN'S FUNCTIONS

Stratified sampling is a sampling technique which gives a reduction in sample variance if the population from which samples are to be made can be divided into sub groups which have variances smaller than the original population. The sub group selection and the number of samples to be made from each sub group must be selected such that the parameters to be determined from the samples are the same for the stratified sampling as for samples taken from the original population. The use of Green's functions to determine the distribution of walk origins on an approximating region R_a contained by R permits the division of the original sample population that exists at ξ to be divided into sub groups with variances smaller than that of the population at ξ. The technique described below was implemented and shown to give a variance reduction before its recognition as an application of stratified sampling.

The technique consists of performing walks from the boundary of the approximating region R_a which contains the point, ξ, for which a solution is sought. The boundary of R_a is divided into M segments, ΔA_i. The proportion, p_m, of the total number of walks, N, to be originated from the mth segment of R_a is determined by

$$p_m = -\int_{\Delta A_m} G_{na}(\xi, s_a)\, ds_a \tag{9}$$

where $G_{na}(\xi, s_a)$ is the normal derivative of the Green's function for R_a. Note that p_m is the probability of a walk originating at ξ intersecting the boundary segment ΔA_m. The segments must be small enough that the solution to Laplace's equation at the mid points of two adjacent segments does not differ greatly.

Consider the solution of Laplace's equation using the technique described above for a region R, Green's

function with normal derivative $G_n(\xi, s)$ and with boundary function $\phi(s) = s$ on a portion of the boundary, Δs, and zero elsewhere. Divide the boundary of the approximating region into M equal segments ΔA_m. For the problem described, the solution is

$$\bar{S} = \frac{1}{N} \sum_{k=1}^{N} S_k \tag{10}$$

where $\bar{S}$ is the average value of the intersection coordinates on ΔS. The variance of the solution is[9] with R_a

$$V(\bar{S}) = \frac{1}{N} \sum_{m=1}^{M} p_m \sigma_m^2 \tag{11}$$

without R_a

$$V(\underset{\sim}{\bar{S}}) = \frac{1}{N} \left(Qu^2 + \sum_{m=1}^{M} p_m \sigma_m^2 + 2 \sum_{m=1}^{M} p_m (u_m - u)^2 - \sum_{m=1}^{M} Q_m u_m^2 p_m \right) \tag{12}$$

giving a variance reduction

$$\delta(V) = \frac{1}{N} \left(Qu^2 + 2 \sum_{m=1}^{M} p_m (u_m - u)^2 - \sum_{m=1}^{M} Q_m u_m^2 p_m \right) \tag{13}$$

where

p_m = the probability of intersecting ΔA_m
σ_m^2 = the variance of the average value of intersection coordinates from walks originating at ξ_m, the mid point of ΔA_m
u_m = the expected value of the intersection coordinates resulting from walks originating at ξ_m
u = the expected value of the intersection coordinates resulting from walks originating at ξ

$$Q = -\int_s G_n(\xi, s)\, ds$$

$$Q_m = -\int_s G_n(\xi_m, s)\, ds.$$

The wavy line under $\bar{S}$ in Equation 12 denotes the fact that in order to get the equation in terms of Equation 11 it was necessary to assume that any walk originating from ξ and intersecting a segment of the boundary of R_a continued its motion from the mid point of the segment. For sufficiently large M the difference in results is negligible. For the experimental verification given below a value of $M = N$ was used.

For verification of the above results, Laplace's equation was solved for the region and boundary values given in Figure 3.

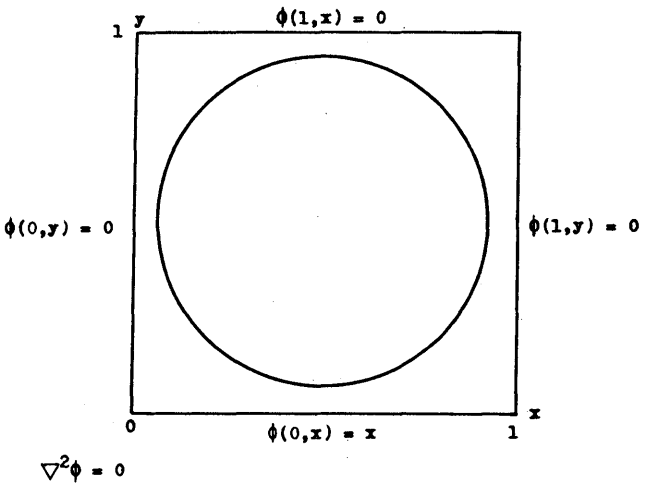


Figure 3

A circle with a radius of .45 centered at (.5, .5) was used for R_a. Solutions were obtained for $x = y = .5$. Approximately 100 walks per second were made—this excludes the time required to compute the number of walks made from each segment.

For each value of N, ten solutions were made. The ten values were used to compute the standard deviation of the sample average. The sample average is the so-

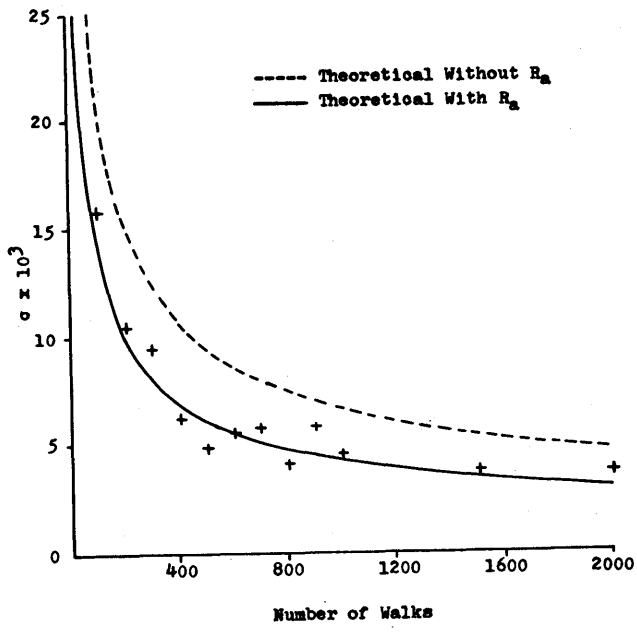


Figure 4—Standard deviation for $X = .5$, $Y = .5$, with R_a

lution to the given problem. The experimental values were plotted along with theoretical values for σ for the cases with and without R_a using the square root of Equations 11 and 12.

The problem was programmed on an EAI 690 Hybrid Computer. The digital portion of the hybrid system computed the initial coordinate values for the walks, the number of walks to be made from a segment, recorded the boundary intersection coordinates, computed the average, and controlled the modes of the analog computer. The analog portion integrated the noise to generate the walk, detected boundary intersections, and performed the filtering for the noise generators.

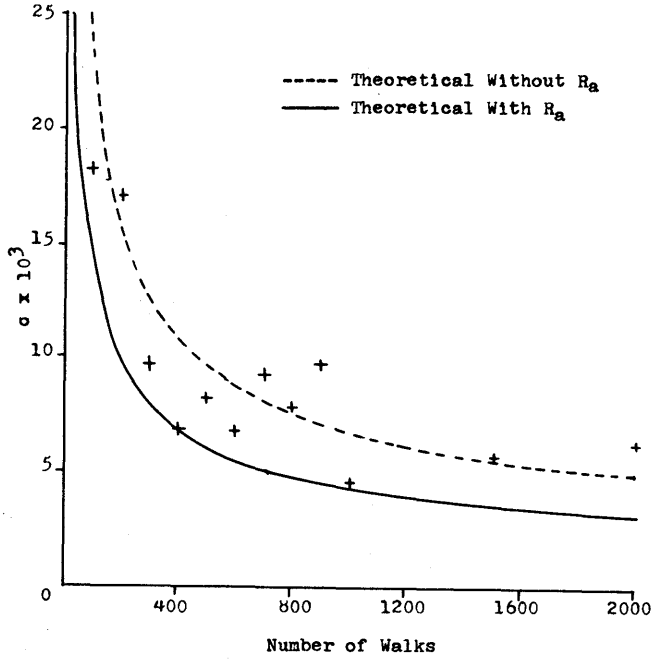


Figure 5—Standard deviation for $X = .5$, $Y = .5$, without R_a

Figure 4 shows the results for $x = .5$, $y = .5$ with R_a. Figure 5 gives the results for the same problem without R_a. The data points would have been closer to the theoretical curves if more than ten solutions had been made at each value of N. However, the improvement in standard deviation using R_a is apparent.

By using R_a for the solution of Laplace's equation the following benefit is realized:

> for $x = y = .5$, a 58% reduction in variance or the same variance as without R_a with 42% as many walks.

For the problem of Figure 3, Figure 6 gives the theoretical change in variance for $x = .5$ and $.05 \leq y \leq .95$ with a radius of .45 and Figure 7 shows the theoretical change in the variance in the sample average as the radius of the approximating circle is varied. Note in Figure 6 that the variance is a function of position and that for a given variance in sample average the number of walks required is not a constant.

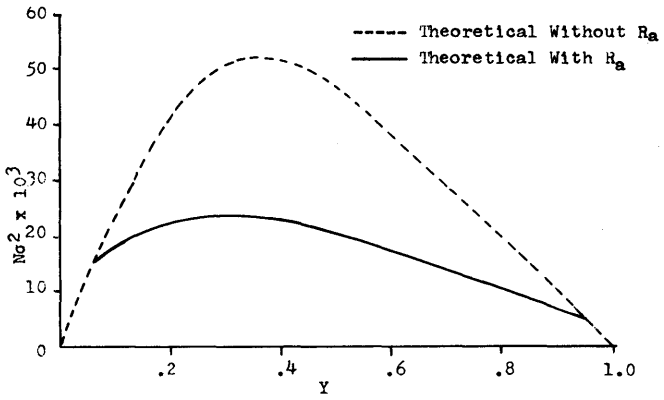


Figure 6—Variance for $X = .5$ using a circular R_a centered at $X = Y = .5$ with a .45 radius

The technique described and demonstrated in this section gives a substantial reduction in error in the continuous random walk solution of Laplace's equation. For a given number of walks the amount of error reduction is dependent on the position of the point for which a solution is sought. The next section contains an extension to the technique which in many cases increases the variance reduction.

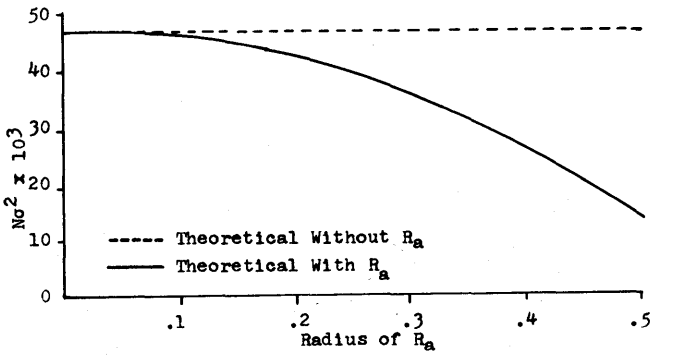


Figure 7—Variance with R_a as a function of radius of circular R_a centered at $X = Y = .5$ for the point $X = Y = .5$

VARIANCE IMPROVEMENT USING BOUNDARY COINCIDENCE

The continuous random walk solution of Laplace's equation with boundary function $\phi(s)$ for a region R with approximating region R_a can be written as

$$\phi(\xi) = \sum_{m=1}^{M} \frac{N_m}{N} \frac{1}{N_m} \sum_{i=1}^{N_m} \phi(s_{im}) \tag{14}$$

or

$$\phi(\xi) = \sum_{m=1}^{M} p_m \phi(\xi_m). \tag{15}$$

$\phi(\xi_m)$ is the sample average at the mid point of the mth segment, N_m is the number of walks intersecting the mth segment of R_a and s_{im} is the coordinate of the intersection of the ith walk from the mth segment.

Equation 15 is the continuous random walk solution of Laplace's equation in R_a with boundary function $\phi(\xi_m)$ on ΔA_m. Since p_m may be computed using the approximating region Green's function, Equation 15 can be written

$$\phi(\xi) = \sum_{m=1}^{M} \left[-\int_{\Delta A_m} G_{na}(\xi, s)\, ds \right] \phi(\xi_m) \tag{16}$$

or

$$= -\sum_{m=1}^{M} \int_{\Delta A_m} \phi(\xi_m) G_{na}(\xi, s)\, ds \tag{17}$$

where G_{na} is the normal derivative of the Green's function for R_a. If R_a can be positioned such that portions of its boundary are coincident with the boundary of R then the boundary function on the coincident portions of R_a need not be approximated by $\phi(\xi_m)$ but is the same as the boundary function for R. Equation 17 can then be written

$$\phi(\xi) = -\int_{s_c} \phi(s) G_{na}(\xi, s)\, ds + \sum_{m=1}^{L} p_m \phi(\xi_m) \tag{18}$$

where s_c is the coincident portion of the boundaries and L is the number of segments not coincident.

The second term of Equation 18 can be written

$$\sum_{m=1}^{L} \frac{N_m}{N} \frac{1}{N_m} \sum_{i=1}^{N_m} \phi(s_{im}) \tag{19}$$

which can be written

$$\frac{1}{N} \sum_{i=1}^{K} \phi(s_i) \tag{20}$$

where s_i is the ith intersection coordinate,

$$K = \sum_{m=1}^{L} N_m, \tag{21}$$

and

$$0 < K \leq N. \tag{22}$$

K is the number of walks actually made and is dependent on the extent of the boundary coincidence and the coordinates of the point for which a solution is sought. N will now be referred to as the base number of walks.

Equation 11 gives the variance when R_a is used which is the weighted sum of the variance from the M segments of R_a. The variance of the coincident segments is zero giving an additional decrease in the variance.

EXAMPLE PROBLEM

The solution to Laplace's equation is desired for the region shown in Figure 8 and boundary conditions:

$$\phi(0, B) = \beta \qquad 0 \leq \beta \leq \beta$$
$$\phi(\beta, \alpha) = B \qquad 0 \leq \alpha \leq A$$
$$\phi(A, \beta) = \beta \qquad 0 \leq \beta \leq B$$
$$\phi(\alpha, \beta) = 0 \qquad \text{elsewhere}$$

where α and β refer to coordinates of points on the boundary.

The Green's function for a rectangle is[10]

$$G(x, y, \alpha, \beta) = \frac{\dfrac{2}{a} \sum_{m=1}^{\infty} \sinh\left(\dfrac{m\pi y}{a}\right) \sinh \dfrac{m\pi}{a} (b - \beta) \cdot \sin\left(\dfrac{m\pi x}{a}\right) \sin\left(\dfrac{m\pi \alpha}{a}\right)}{\dfrac{m\pi}{a} \sinh \dfrac{m\pi b}{a}}$$

for

$$0 \leq y \leq \beta \tag{23}$$

$$= \frac{\dfrac{2}{a} \sum_{m=1}^{\infty} \sinh\left(\dfrac{m\pi \beta}{a}\right) \sinh \dfrac{m\pi}{a} (b - y) \cdot \sin\left(\dfrac{m\pi x}{a}\right) \sin\left(\dfrac{m\pi \alpha}{a}\right)}{\dfrac{m\pi}{a} \sinh \dfrac{m\pi b}{a}}$$

for

$$\beta \leq y \leq b$$

where

$$0 \leq x \leq a \qquad 0 \leq \alpha \leq a$$

$$0 \leq y \leq b \qquad 0 \leq \beta \leq b.$$

The coordinates of the point being solved for are (x, y).

Equation 23 can also be expressed as a Fourier series in y by interchanging a and b, x and y, and α and β. Both representations give the same results but, to obtain faster convergence the Fourier series in x should be used at points for which

$$\left(\frac{y-\beta}{a}\right) > \left(\frac{x-\alpha}{b}\right) \tag{24}$$

and the Fourier series in y when the reverse of Equation 24 is true. The exponential form of Equation 23 was used for developing the digital representations of the equation.

Using the Green's function for a rectangle and the two approximating regions 1234 and 4567 shown in Figure 8, the problem was programmed on an EAI 690 hybrid computer. The digital-analog problem split was as follows:

The digital computer calculated which of the two approximating regions would require the least random walks, the contribution the coincident boundary portions made to the solution, and the number of walks to be made from each non-coincident segment. In addition, the digital computer controlled the modes of operation of the analog computer, monitored sense lines signaling boundary intersections, stored the boundary intersection coordinate values, computed the proper boundary function values, and set the initial conditions for the x and y integrators.

The analog computer performed the random walks by integrating the noise inputs, provided shaping filters for the noise, and with analog comparators and parallel logic components simulated the boundary and flagged the digital computer when an intersection occurred.

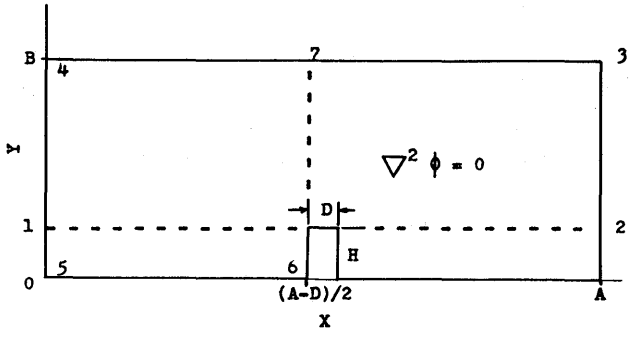


Figure 8—Region for which solution to Laplace's equation is desired

TABLE I—Solution for Laplace's Equation for Region of Figure 8 with Base Walk Number of 1000

	SOLUTION USING R_a			1000 Walks WITHOUT R_a		
Y	DIGITAL	HYBRID	|ERROR|	#WALKS	HYBRID	|ERROR|
			$X = 100$			
36	35.33	35.19	.14	49	35.60	.28
32	30.62	30.54	.08	97	31.56	.94
28	25.83	25.89	.06	143	26.52	.31
26	23.38	23.55	.17	161	24.08	.70
24	20.89	20.72	.17	178	20.56	.33
22	18.33	18.25	.08	190	18.76	.43
20	15.69	15.63	.06	212	15.76	.07
18	12.93	13.20	.27	201	13.00	.07
16	10.02	9.89	.13	191	10.76	.75
14	6.90	7.11	.21	154	8.20	1.30
12	3.54	3.42	.12	91	4.48	.94

Table I contains the results for the region shown in Figure 8 with dimensions:

$$A = 200$$

$$B = 40$$

$$D = 10$$

$$H = 10,$$

a 5-unit boundary segment for region 1234, a 2-unit segment for region 4567, and a base walk number of

TABLE II—Solution for Laplace's Equation for Region of Figure 8 with Base Walk Number of 10,000

SOLUTION USING R_a				
Y	DIGITAL	HYBRID	| ERROR |	# WALKS
		$X = 100$		
36	35.33	35.34	.01	490
32	30.62	30.61	.00	970
28	25.83	25.71	.12	143
26	23.38	23.40	.02	1610
24	20.89	20.82	.07	1780
22	18.33	18.33	.00	1900
20	15.69	15.66	.03	2120
18	12.93	12.99	.06	2010
16	10.02	10.02	.00	1910
14	6.90	6.93	.03	1540
12	3.54	3.61	.07	910

1000. The solutions both with and without an approximating region are compared to the results obtained using finite differences and a relaxation type numerical solution.

Table II gives the results using an approximating region and a base walk number of 10,000. Solutions for many other points were made[11] with similar reductions in error. Note in Tables I and II that less than 25% of the base number of walks is actually made.

QUICK LOOK CAPABILITY

One advantage of the continuous random walk with or without R_a is the ability to find the solution for just one point in the region while the solution by conventional digital techniques requires the solution over the whole region to obtain the solution at a particular point. Using the continuous random walk technique a solution for ϕ can be found at a point or points as the geometry is changed allowing a quick look at the effect of a geometry change at some critical point. For the problem of the preceeding section Figure 9 shows the resultant change in ϕ at the point $x = 100$, $y = 20$, as H varies from 0 to 20 with a segment increment of 5 and 1000 for the base number of walks. The data for the curve was taken in less than 15 minutes, the time required for one conventional digital solution.

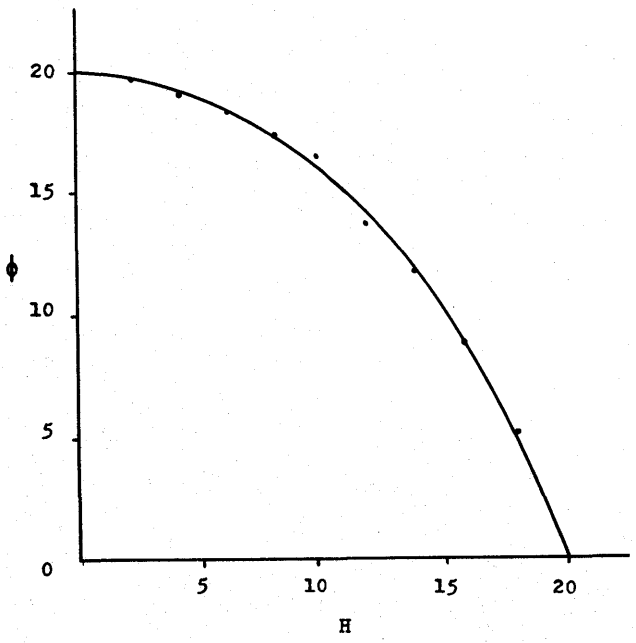


Figure 9—Solution of Laplace's equation for $X = 100$, $Y = 20$ in the region of Figure 8 as H is varied

SOLUTION OF POISSON'S EQUATION WITH CONSTANT FORCING FUNCTION

Handler[12] proposed solving Poisson's equation with constant forcing function by making use of the fact that the average time, T, for a random walk satisfies the equation

$$D_1 \frac{\partial^2 T(x,y)}{\partial x^2} + D_2 \frac{\partial^2 T(x,y)}{\partial y^2} = -1 \qquad T(s) = 0 \quad (25)$$

and that the solution for Poisson's equation

$$\frac{\partial^2 \phi(x,y)}{\partial x^2} + \frac{\partial^2 \phi(x,y)}{\partial y^2} = -\lambda \quad \phi(s) = f(s) \quad (26)$$

can be found as the sum of the solutions of

$$\frac{\partial^2 \phi_1(x,y)}{\partial x^2} + \frac{\partial^2 \phi_1(x,y)}{\partial y^2} = 0 \quad \phi(s) = f(s) \quad (27)$$

and

$$\frac{\partial^2 \phi_2(x,y)}{\partial x^2} + \frac{\partial^2 \phi_2(x,y)}{\partial y^2} = -\lambda \quad \phi(s) = 0 \quad (28)$$

where $\phi_2 = \lambda T$. Random walks were performed with a value proportional to the average walk time being added to the resulting average of the intersected boundary values.

The technique described in the previous section is also applicable to the solution of Poisson's equation with constant forcing function.

Using the Green's function, Equation 28 can be solved for $R_a{}^3$ giving the average time for a random walk to the boundary of R_a.

The solution for Poisson's equation is then

$$\phi_P(x,y) = \phi_L(x,y) + \lambda T(x,y) + \frac{\lambda D}{N} \sum_{i=1}^{K} T_i \quad (29)$$

where $\phi_L(x,y)$ is given by Equation 18, $\lambda T(x,y)$ is given by $\phi_2(x,y)$ of Equation 28, and the last term is the average time contribution from the walks actually made from the non-coincident portion of the boundary of R_a. D is the power spectral density of the noise source. N is the base number of walks and K is the actual number of walks performed.

Poisson's equation with λ equal to 1 was solved for the region and boundary values shown in Figure 10. Two approximating rectangles were used,

$$0 \leq x \leq 1.0$$
$$0 \leq y \leq .9$$

and

$$0 \leq x \leq 1.0$$
$$0 \leq y \leq .6.$$

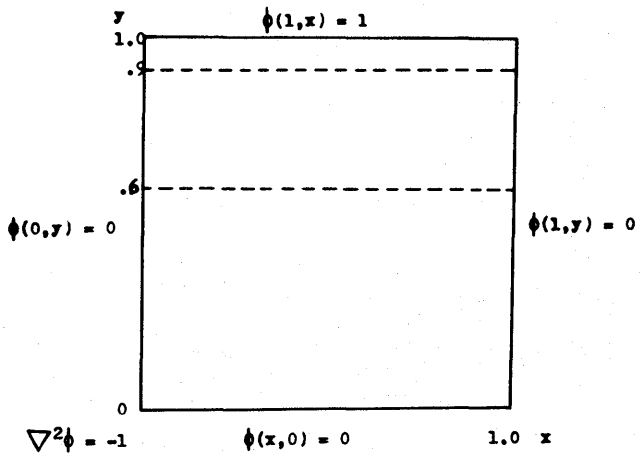


Figure 10—Rectangular region for which a solution to Poisson's equation is desired

The solution for $x = y = .5$ is given in Figure 11. A base number of 1000 walks was used. Approximately 100 walks were performed per second.

The time required for the K walks actually made was measured by counting a one-hundred kH_z clock pulse gated to a counter by the operate pulse supplied to the analog computer by the digital computer. The terms of Equation 29 were then computed separately and added to get the plotted result. The total solution could have been obtained by a single program (memory capacity permitting) since an interval timer which can be set and checked by FORTRAN statements is provided as part of the interface. However the objective was just to demonstrate the applicability of the technique to the problem so separate programs already written and capable of computing the terms of Equation 29 were used. Primary objective of the research for this paper was to develop a method of solution for Laplace's equation.

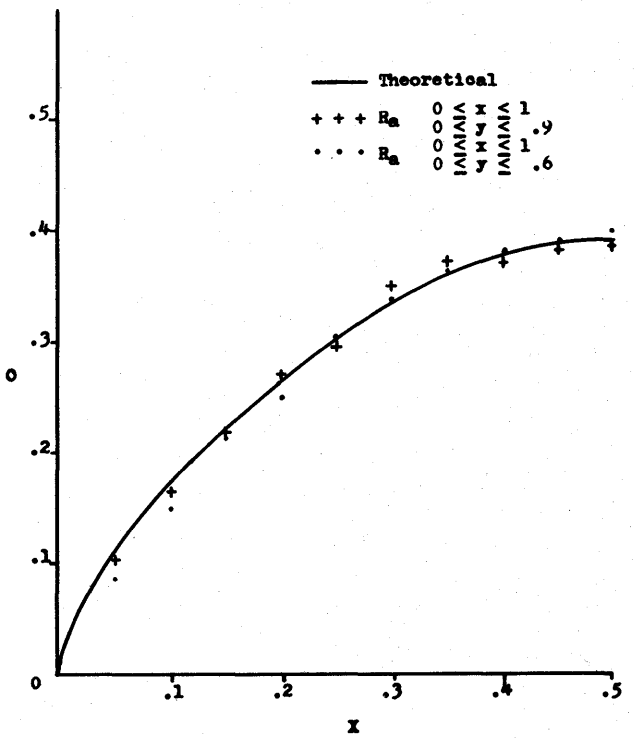


Figure 11—Continuous random walk solution of Poisson's equation using R_a for $Y = .5$

NOISE GENERATORS

Fundamental to the success of the continuous random walk technique is a stable noise generator with known statistical parameters. Initial work reported in this paper was attempted with a maximum length shift-register pseudo-random noise generator. The band width for which the amplitude distribution was Gaussian was too narrow so the data presented were taken using a Burr Brown 4006/25 noise generator which proved satisfactory. Recent work[13] on non-maximal length shift-register pseudo-random noise generators suggests that sufficient band width can be obtained with these stable, predictable generators.

CONCLUSIONS

The goal of the research for this paper was to develop a technique which would give decreased variance in the continuous random walk solution of Laplace's equation. The goal was attained using a stratified sampling technique.

The technique was applied to problems for which analytical solutions could be obtained and was shown to give theoretically predictable variance reductions.

A method for applying the technique to the solution of Poisson's equation with constant forcing function was developed.

Solutions to problems using continuous random walks may be too inaccurate for some applications but the ease with which boundaries may be altered and the ability to quickly examine the effects of boundary alteration on a few critical points makes the solutions an economical prelude to a more precise analysis on a digital computer.

REFERENCES

1 K CHUANG L F KAZDA T WINDERNECHT
A stochastic method of solving partial differential equations using an electronic analog computer
Project Mich Report 2900-91-T Willow Run Labs University of Michigan June 1960

2 M C WANG G E UHLENBECH
On the theory of the Brownian Motion II
Rev Mod Phys Vol 17 pp 323-342 1945

3 I PETROWSKY
Uber des irrfahrtproblem
Math Ann Vol 109 pp 425–444 1934

4 W D LITTLE
Hybrid computer solutions of partial differntial equations by Monte Carlo methods
PhD thesis University of British Columbia October 1965

5 H Handler
Monte Carlo solution of partial differential equations using a hybrid computer
PhD Thesis University of Arizona

6 A W MARSHALL
An introductory note
Symposium on Monte Carlo Methods Edited by H A Meyer Wiley Publications 1954

7 B FRIEDMAN
Principles and techniques of applied mathematics
Chapter 3 Wiley 1956

8 J KEILSON
Green's function methods in probability theory
Hafner Publishing Company 1965

9 E L JOHNSON
A variance reduction technique for the continuous random walk solution of Laplace's equation
PhD Thesis University of Kansas 1969

10 Op Cit FRIEDMAN p 267

11 Op Cit JOHNSON

12 Op Cit HANDLER

13 L RUNYAN
Non-maximal length shift-register pseudo-random noise generators
Master's Thesis Wichita State University 1970

APPENDIX A

The equivalence of the Green's function for a region and the probability density function for boundary intersections

The negative of the normal derivative of the Green's function for Laplace's equation in region R is used as the probability density function for boundary intersections. A heuristic argument is presented below to justify this use.

Consider the region shown in Figure 1A. Wang, et al, have shown that Equation 1A converges with probability 1 to the value of the solution to Equation 2A

$$V(\xi) = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} f(s_k) \tag{1A}$$

$$\nabla^2 V(\xi) = 0 \qquad V(s) = f(s) \tag{2A}$$

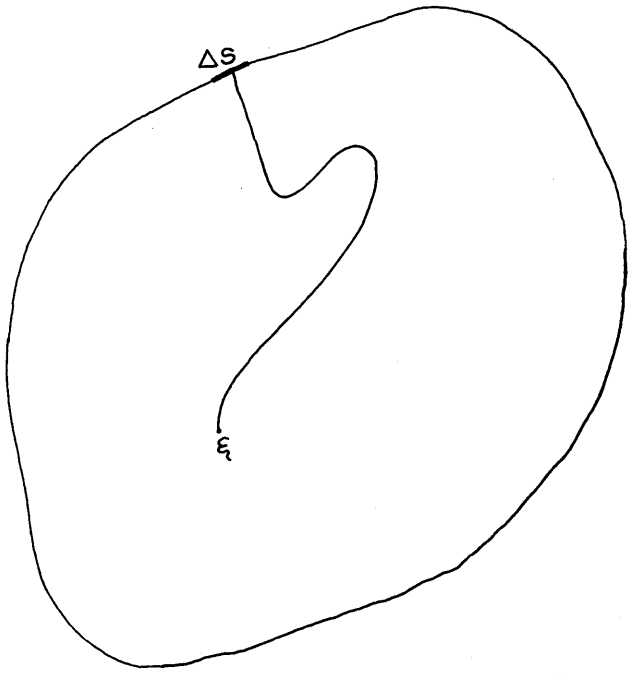


Figure 1A

where ξ represents the coordinates of the walk origin and s_k is the coordinate of the boundary intersection.

Assume that the Green's function $G(\xi, s)$ is known and that the function $f(s)$ is defined as

$$f(s) = 1 \text{ on } \Delta s$$
$$= 0 \text{ elsewhere.} \tag{3A}$$

The solution of Equation 3A is then

$$V(\xi) = -\int_s f(s) G_n(\xi, s)\, ds = -\int_{\Delta s} G_n(\xi, s)\, ds. \tag{4A}$$

For $f(s)$ equal to Equation 3A, Equation 1A becomes

$$V(\xi) = \lim_{N\to\infty} \frac{M\Delta_s}{N} \tag{5A}$$

where $M\Delta_s$ is the number of walks which intersect Δs. Equation 5A gives the probability of a random walk originating at ξ intersecting the boundary segment Δs. Comparing Equations 4A and 5A, it is seen that the solution to Laplace's equation for the given region and boundary function is the probability of a walk originating from ξ intersecting the boundary on Δs. Therefore if we define a random variable S which takes on values equal to the boundary coordinate of the inter-

section we can write

$$p(s_1 \leq S \leq s_1 + \Delta s) = \int_{s_1}^{s_1+\Delta s} - G_n(\xi, s)\, ds. \quad (6A)$$

Further, if $f(s)$ is defined by

$$f(s) = 1 \quad (7A)$$

then

$$V(\xi) = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} 1 = 1 \quad (8A)$$

which gives the obvious value of 1 for the probability of intersecting the boundary. Using Green's function we can write

$$V(\xi) = \int_{s} - G_n(\xi, s)\, ds = 1. \quad (9A)$$

By comparing Equations 8A and 9A the property that the integral of a density function over the range of its variable results in a value of 1 is demonstrated. The probability density properties demonstrated by Equations 6A and 9A are used in this paper for the development of a variance reduction technique for continuous random walks.

Design of automatic patching systems for analog computers

by T. J. GRACON

Control Data Corp.
Sunnyvale, California
and
J. C. STRAUSS

Carnegie-Mellon University
Pittsburgh Pennsylvania

INTRODUCTION

Continued use of analog computers depends heavily on the discovery of ways to ease programming difficulties. Many of these difficulties are caused by the existence of the analog patchboard and the necessity of hand patching every problem which is a very time consuming and error prone process. To save the program for future use involves storing the wired patch panel, requiring that each user has his own patch panel and wiring set. Providing these to each user is very costly. The alternative to allowing each user to have his own patchboard requires that the program must be taken off the board and repatched every time a program run is desired. This too is costly since it requires duplication of the initial set up and debugging effort.

One often proposed method of eliminating these difficulties involves the building of a programmable switching matrix. This automatic system would process a problem stated in some standard form, identify the set of switches in the matrix needed to interconnect the required set of analog components in the problem specified configuration, and cause the close of these switches to "patch" the problem. Because of the speed of digital switching and decoding devices, the setup time of each problem would be reduced enormously. Storing of the analog program then becomes a matter of storing only a digital command sequence which can be done on inexpensive paper tape or cards.

In addition, the existence of an automatic patching system on an analog computer creates a new computing environment that offers possibilities for major changes in the use of analog machines. Batch processing becomes a distinct possibility. While one problem is being run on the analog, the digital controller can be processing the next problem in preparation for setting it on the switching matrix. This foreground-background mode of operation should increase throughput significantly. The system can possibly be coupled as shown in Figure 1, with a hybrid programming language such as described in Reference 1. This problem oriented language will take a set of differential equations and from it generate a complete, properly scaled, analog patching diagram. The combined system makes analog programming very similar to standard digital computer programming and, as a result, provides a new computing tool (the analog) to the approximately 80% of all scientific computer users with digital experience only.

OVERVIEW OF THE AUTOMATIC PATCHING SYSTEM DESIGN PROBLEM

The simplest design of the switching matrix would allow every component output to connect to every component input. Such a system could provide every conceivable connection asked of it. Unfortunately, the number of switches needed to implement this sample system tends to be so large that the cost of the switching matrix would far exceed the cost of the computer itself. For example, there are $\simeq$200,000 valid connections between analog components on the EAI 680. At \$1.50 a switch, the cost of switches alone would be \$300,000.

The economic constraint of requiring that the unit be moderately priced generates the problem that is considered in this work. The problem is basically to design a system that:

(1) Allows any component output to component input

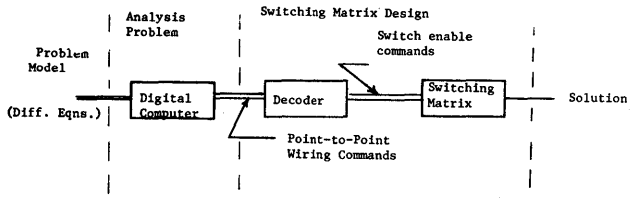


Figure 1—A coupled automatic analysis—Automatic patching system

connection that might be required by an actual analog problem.

(2) Allows a sufficient number of connections to be made simultaneously so that significant class of actual analog problems can be solved.

(3) Generates the interconnections automatically with no interaction required by the user except to input initial data,

(4) Provides 1, 2, and 3, above at reasonable cost.

This total systems cost contains the hardware cost of the switching matrix and its associated activating system, the cost of designing and developing the control programming required to operate the system, the labor cost for doing the hardware design, and some measure of the price being paid to run problems on operating systems. Experienced designers realize that tradeoffs exist between these sections of the design. Putting most of the design effort into one area can reduce its associated cost but tends to raise the complexity of the other sections and leads to a higher priced system. A good design tries to consider the interaction between these design sections and attempts to reduce the total systems cost by evenly distributing the design effort over the entire system. The remainder of this paper considers the entire systems design and proposes a design procedure that produces an efficient, economical, operating system.

DESIGN OF AN AUTOMATIC PATCHING SYSTEM

The automatic patching system is being designed to replace the process of hand patching. It follows then that the operations to be performed by the automatic system can be identified by noting the basic steps involved in hand patching. The user first translates his set of equations into a block diagram representation of the problem. A 3rd order differential equation and block diagram is shown in Figure 2. The block diagram begins to show a dependence on the particular computer being used since the need to scale the problem often requires that additional components be added. Each element of the block diagram is then associated with a particular component on the analog. This process is called allocation in the remainder of the paper. The block diagram of Figure 2 is allocated to a small computer in Figure 3. Once the allocation is performed, the specific points on the patch panel that must be interconnected are known. Connecting these points together by inserting patch cords into the proper holes completes the hand patching.

The automatic system must include the allocation of the problem of the board. As was mentioned earlier, this fixes the set of input–output connections to be provided by the switching matrix. If the allocation program is not incorporated into the matrix design, problems may be allocated in such a way that the matrix cannot provide the needed connections and the system fails. Automatic scaling and block diagram generation are more properly left to the hybrid language[1,2] mentioned earlier.

The next operation by the automatic systems involves determining the set of switches in the matrix which, when closed, connect the specified inputs to their respective outputs. The difficulty of this switch determination, or pathrouting, problem is, of course, dependent on the configuration of the switching matrix and must be considered in the systems design. Later in the paper the problem is more closely studied.

To date, investigators[3–6] have begun their designs

$$\dddot{x} + a\ddot{x} + b\dot{x} + cx = 0$$

$$\ddot{x}(o) = k$$

$$\dot{x}(o) = o$$

$$x(o) = o$$

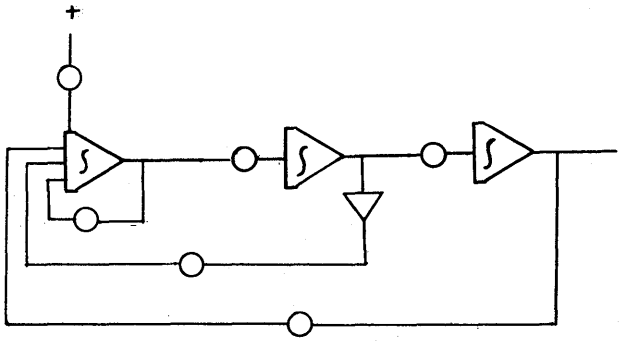


Figure 2—Block diagram of 3rd order O.D.E.

by trying to determine the best switch configuration for their own computers. A reader of these works quickly realizes that many potentially usable configurations exist and faces the problem of selecting among them. What is obviously wanted is the configuration that provides the needed capacity at minimal cost. If a curve can be drawn for each system plotting the cost of the system vs. capacity (see Figure 4) then the decision can be made by determining the capacity required by the user and simply selecting the least expensive system for that capacity.

Many questions quickly arise, the most obvious being how is capacity measured? What happens if the least expensive system is the most difficult on which to perform the allocation and path-routing of problems? Can a cost vs. capacity relation be found for each considered system?

The answer to the first question is found from considering the function of the switching matrix, namely, to provide a certain number of connections between matrix inputs and matrix outputs. The capacity of a system is the number of simultaneous input/output connections it can provide. It then follows that the cost vs. capacity curves for each system can be alternately expressed in terms of cost as a function of the number of inputs, outputs, and connections required. The actual functional relation depends on the organization of the considered system. Figure 5 presents

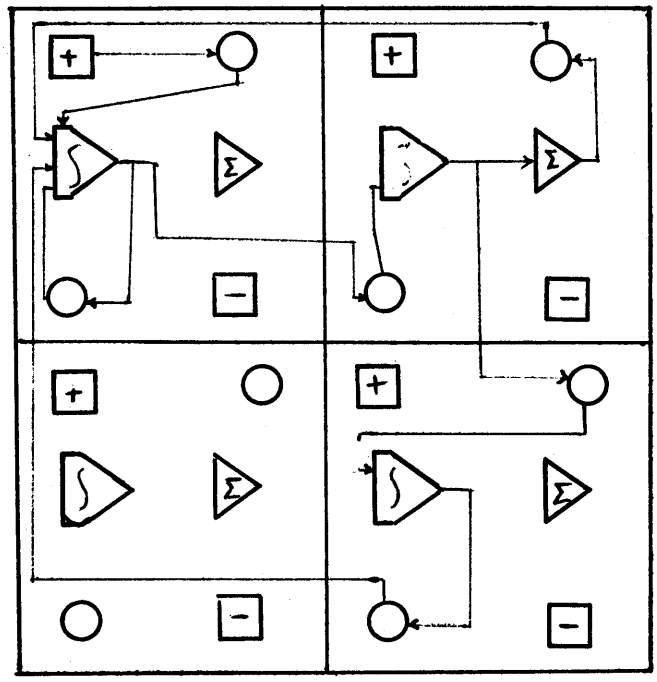


Figure 3—Allocation of 3rd order O.D.E. to a small computer

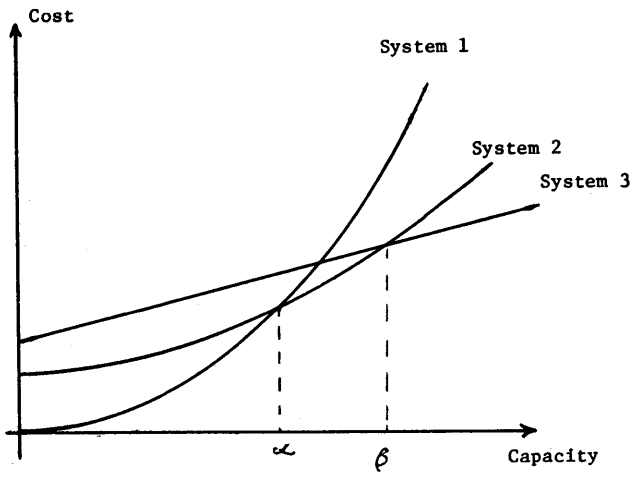


Figure 4—Cost = number of switches

an example of determining a cost function for each system. Reference 6 develops these functions for a variety of systems and indicates that such functions can generally be found. The allocation and path routing problems can be imbedded in the systems selection by considering them when determining the number of needed connections for each system. The way they should be considered is explained in the remainder of the text.

The real system design problem exists only when the designer admits to the requirement for a multilevel switching matrix organization. As with most complex systems, progress can be made only when the system is divided into subsystems of reasonable conceptual size. In addition, the tendency for the number of switches to increase as the square of the number of components indicates a significant cost savings can be realized by sectioning the system into smaller blocks of components with some means provided to interconnect these blocks. In this case multilevel means that conceptually all possible connections are separated into different types (levels). An individual switching matrix is considered for each level. The switching organization for each level may or may not be the same. An example clarifies this. An experienced analog user realizes that most connections tend to be made between elements located in close proximity to each other. Only a few connections "reach" across the board. This strongly suggests a two level system, one level to provide connections locally within a small group of components (normally this group is called a module), and one level to provide connections globally between these groups (this level is

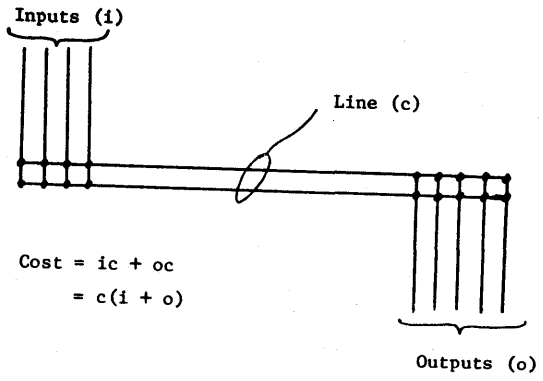


Figure 5—Cost as a function of inputs, outputs, and connections for a concentrator—Expander system

called the intermodule system). This is in fact the system proposed in References 3–8 and will be the basis of the design implemented in this paper.

Now a real design problem exists since the designer can share the required capacity over the two levels in some manner. This provides the designer with the means of generating a cost/effectiveness tradeoff via parameters under his control. The following definitions help to quantify this idea.

Let

$T_{k,l}$ = total cost system using organization (k) intra-module and organization (l) inter-module
C_k = cost of intramodule system (k)
C_l = cost of intermodule system (l)
o_k, i_k, c_k = outputs, inputs, and connections of intra-module system (k)
o_l, i_l, c_l = outputs, inputs, and connections of inter-module system (l)

Then

$$T_{k,l} = C_k + C_l$$
$$= F_{k,l}(o_k, i_k, c_k, o_l, i_l, c_l) \quad (1)$$

The 6 variables of $F_{k,l}$ are known to be related because the sum of the capacities of each level is equal to the total required capacity. The multilevel design then consists of determining the natural relations that exist among these variables (which includes proper allowance for the allocation and path routing problems). Once these relations are known, the cost function can be expressed as a function of one variable, a, (the meaning of, a, will become clear later):

$$T_{k,l} = T_{k,l}(a)$$

Reducing the cost functions to single variable functions makes the minimization easier to intuitively visualize. Experience, and the discussion to follow supports this approach. The design can be stated as an optimization process in which the objective is to:

$$\underset{k,l,a}{\text{minimize}} \quad T_{k,l}(a)$$

This is shown graphically in Figure 6. Simply speaking this means pick the lowest cost system. Where this system reaches its minimum is the optimal value of "a" which trough the relations above leads to the best values of the 6 variables in Equation. Knowing the system (k, l) and values of those variables in effect completes the design.

Phase I of the design—Modular relations

Accumulated experience in analog computer programming has shown that components tend to be interconnected in certain basic patterns. It is unnecessary for every component to directly connect to every other component in that the probability of specific connections decreases rapidly with increasing intercomponent distance. Thus, by physically locating together the components that are most likely to be used together, programming efficiency will be increased. These groupings of components are called modules. The component composition of the modules is critical to the design. All proposed designs to date have been built around symmetric modules. Attempts at designing around non-identical modules have failed to produce a design that can handle more than a limited class of problems.[3]

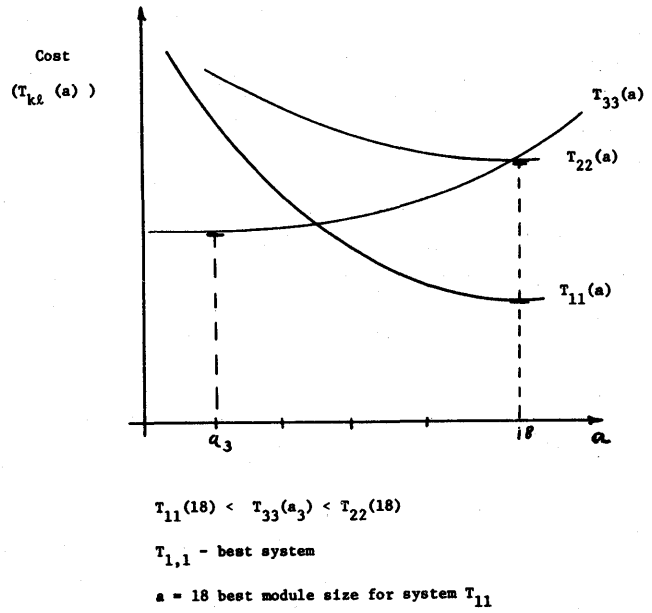


Figure 6

Within the framework established in this paper, the decision to make all modules have identical component composition (be symmetric) proves to be a wise one. Knowing that all modules are identical means that the number of switching matrix inputs and outputs, on both levels, can be expressed as a function of one variable, (the number of modules, a,) above. This is not hard to see. Dividing the component complement by the number of modules (only those values which lead to integer results make sense and are allowed) gives the complement/module. The number of inputs and outputs of this set of elements is known from observing them on the computer. These two facts make it possible to determine the number of inputs and outputs to the intramodule switching matrix as a function of, a,. The number of inputs and outputs to the intermodule system depends on the number and type of components in a module, and the number of modules, all of which can be expressed explicitly as a function of the number of modules. Reference 6 does this for a large number of systems. The net effect is to reduce the total cost function of Equation 1 from 6 variables to 3.

$$T_{k,l} = g_{k,l}(a, c_k, c_l) \tag{2}$$

Phase II of the design—Relating the number of connections needed to the number of modules

Once the number of connections (c_k, c_l) are related to the number of modules on the computer, Equation 2, can be reduced to a function of one variable and the cost function minimized. These connection numbers (c_k, c_l) specify the capacity needed in the intra- and intermodule systems. A measure of this capacity should be discoverable by analyzing a particular users needs. Specifically if a user provides a set of problems which typify the connection requirements of his particular computing work, then these typical problems can hopefully be analyzed in some sensible way to yield $c_k(a)$, $c_l(a)$. (This does not imply that there will be no similarity between the needs of different users. Perhaps analysis of typical problem groups submitted by a variety of users will indicate that a single system organization, or a few basic ones, can suffice for all situations. Since the automatic patching system problem is still in its infancy this analysis hasn't yet been done. As a result, it seems reasonable to begin by considering one user at a time, instead of a larger group).

Earlier it was indicated that, after a problem is allocated to the board, all connections are known. Thus allocating a problem to a board with a modular organization generates a distribution of connections within and between modules. This distribution can be found by simply counting the inter- and intramodule connections. If the same problem is again allocated to the board now divided into a different number of modules, the connection statistics become dependent to some degree on the number of modules. Allocating each problem to the board under all possible modular sectionings produces a set of data that relates the number of connections needed to the number of modules used. The method of determining this relation is discussed next.

The method of allocating the problems to the board should now be clearly specified. A given problem can be allocated to the patch board either by hand or automatically through an allocation program. In an operating automatic patching system the allocation will be done automatically, implying that any allocation generated statistics used in the design should result from automatically allocating the typical problems. If the switching matrix was designed around hand allocation statistics, the system might fail to handle problems that are automatically allocated to it unless the hand and automatic allocation schemes are remarkably similar.

Intuitively, the allocation algorithm should attempt to put a problem on the board in a way that minimizes the number of intermodule crossovers. The reason for this is simply that if a connection is kept within a module, the number of points to which it may connect is significantly smaller (and thus less costly in switches) than the number of points on the remainder of the board. The algorithm should also attempt to keep the problem in as small a fraction of the board as possible to allow the remainder of the board to be used for some other purpose. The generation of such an algorithm is considered in the next section.

Once the allocation algorithm is implemented it can be applied to the typical problems. For each problem the maximum number of connections within a module and the largest number of inputs or output/module for each modular sectioning of the board is recorded. A worst case loading problem is synthesized from these tables by selecting the largest value of each parameter over the problem set. Figure 7 shows an example taken from Reference 7. The first line says simply that when

a	c_k	c_ℓ/module
18	14	5
9	30	6
6	37	8

Figure 7—Example of a worst case problem

the computer was divided into 18 modules the largest number of connections needed (over all typical problems) within a module was 14 and the largest number of inputs and outputs/module was 5. References 6,7 further explain this worst case problem concept.

The worst case problem relates in a tabular way, the inter- and intramodule connection capacity required to the number of modules. Equation (2) can now be evaluated for all values of c_k, c_l at allowable values of a. The minimum cost system is selected. The value of, a, where the minimum occurs is the optimal module size. Knowing the number of modules specifies the module size and component composition. The values of c_k and c_l are determined by using the values in the worst case problem that correspond to the selected number of modules.

Completing the design—Concerns about routing

The design is nearly complete. The least costly system for each level and its critical design parameters have been identified. Allowance has been made for the difficulty of implementing the automatic allocation by implementing the allocation algorithm first, then providing sufficient capacity in the switching matrix to guarantee that the two will operate properly when combined. Until this point no particular effort has been made to incorporate the path routing problem into the design process. The path routing problem results from using a switching configuration that provides anything other than exhaustive interconnection between inputs and outputs. The problem is completely dependent on the switching configuration used and, as a result, cannot be considered in general (as was the allocation program). Each system must be examined individually to determine if an effective routing algorithm can be found for it. The way to proceed is obviously to list the systems in order of increasing cost. The least expensive system is examined first. If no workable, or easily implementable, system is discovered in a reasonable time, then the next least expensive system is considered. The difference in costs of the two systems (a.e. $T_{kl} - T_{ij}$) gives the designer some feel for whether it is worthwhile to spend more time examining the cheaper system or to try finding the routing algorithm for the next system. In general, a more costly (more switches used) system, since it is providing the same capacity with more switches, offers more ways of possibly connecting inputs to outputs. This implies that the routing problem becomes easier to solve as higher cost systems are considered. These ideas are quantified through the discussion of a particular switching organization in the following section.

CONSIDERATIONS

Hardware configurations

The early works[3–5,8] on automatic patching began with an a priori selection of the switching configuration to be used. All authors sought to minimize the cost of the system by selecting the critical design parameters, such as number of modules and connection load on the inter- and intra-module systems, in some intuitive way. Reference 6 admits that the selection of the configuration to be used on each level should be a variable in the design process. In Reference 6 the required capacity is measured by abstracting a worst case problem from a user specified group of typical problems. The worst case problem enables the relating of all design variables to one variable, the number of modules, and allows the selection of the least expensive system in the manner described in the preceding section of this paper. Unfortunately, the worst case problem was generated by hand patching the typical problems instead of by automatic allocation. Reference 7 and this paper are the first known that actually incorporate the allocation program into the hardware configuration selection and design.

The Allocation program

It is possible to formulate the component allocation task as an optimal resource allocation problem where the goal is to minimize the number of intermodule connections. Binary programming or branch and bound algorithms can be adapted to solve the problem. Unfortunately the combinational size of the problem is so overwhelming that estimates of solution times for even a small-medium size analog[7] indicate that this approach is impractical. No one could rationally propose typing up a computer for hours to do a task that can be done by hand in a few minutes. The original goal of producing a system that is inexpensive to run as well as to build translates into a constraint of requiring that all automatic processing (allocation, path routing) requires only a short (1–5 min.) run time.

This constraint can be met by posing a heuristic allocation algorithm, such as shown in Figure 8. This algorithm attempts to simulate the process of hand allocation. The algorithm has been implemented in Fortran IV and run on Carnegie-Mellon University's Univac 1108 Computer. The implementation for the model computer of References 6,7, takes approximately 5 secs. to compile and has allocated problems of up to 120 components in less than 2 secs. Reference 7 presents a comparison of the allocation of a few problems by

hand and by the computer program. In general, the algorithm works reasonably well. The algorithm is designed to cluster components around integrators, attempting to place most of the connection load on the intramodule system. This effect tends to minimize the number of intermodule crossovers. A comparison of the worst case problems abstracted from the hand generated and program generated statistics is shown in Figure 9.

Future work in automatic patching systems should include serious efforts towards finding the best possible allocation algorithm.

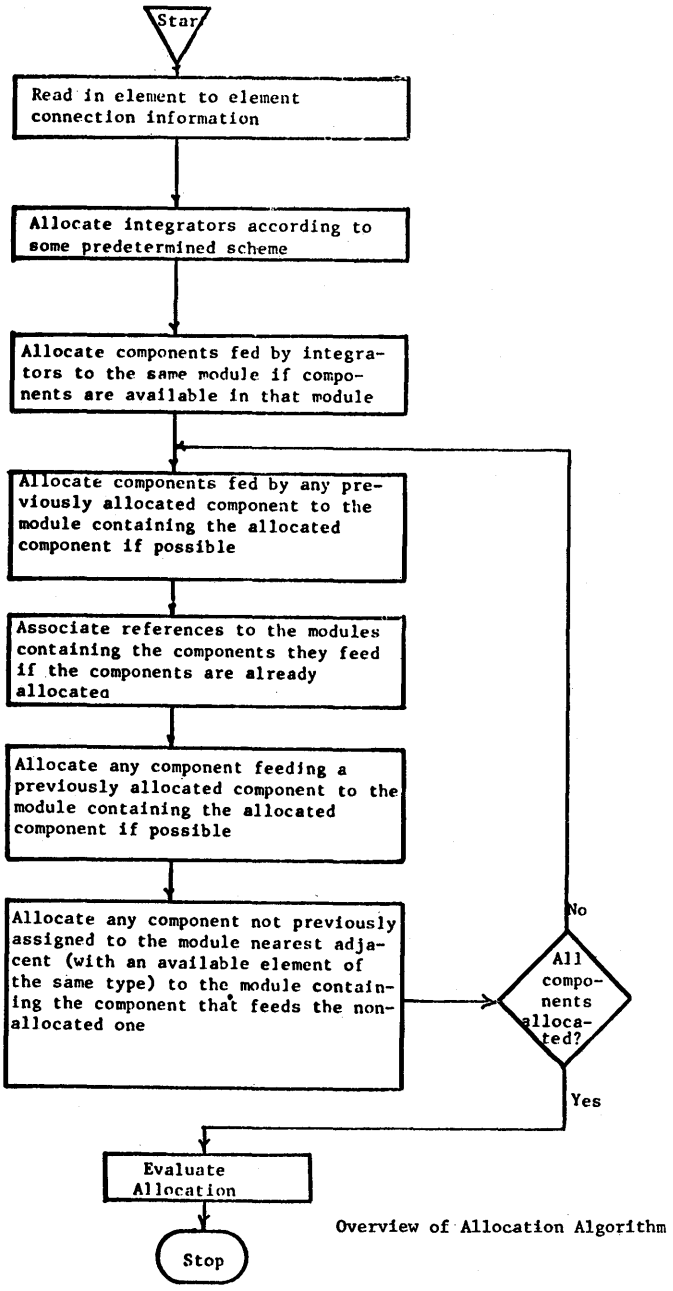
Worst Case Problem

Number of Modules	Maximum Number of Outputs (Inputs) per Moduel		Maximun Number of Connections Within a Module	
	H	P	H	P
18	4	5	13	14
9	6	6	25	30
6	6	8	31	37

H - Hand allocated

P - Program allocated

Figure 9



Overview of Allocation Algorithm

Figure 8—Overview of allocation algorithm

Routing problem

In (7) it is found that the least expensive system occurs when the model computer is divided into the largest permissible number of modules. Because of the modules small size, it turns out to be cheaper to exhaustively interconnect all components within the module (deleting all forbidden connections like pot to pot) than to use any other intramodule switching organization. Since a single switch is uniquely associated with each potential connection within a module no routing problem exists. For any intramodule connection specified by the allocation of a problem, a simple table look up routine will identify the needed switch.

The intermodule routing problem is more difficult to solve. The least expensive system considered depends heavily on the distribution of intermodule connections. For some distributions no path routing can be found. At this time no way is known to practically imbed any distribution constraints in the allocation program. As a result this first intermodule system was considered to be unworkable. The next least expensive intermodule system considered is shown in Figure 10. The symbol

(for concentrator) corresponds to an array inputs of which are the outputs of all components in the module and the output is a bus line which connects to one input point in every other module. These input points (expanders)

allow the module input signal to be connected to any analog component input. The main characteristic of this system is that each concentrator in a module (each output port) connects to one

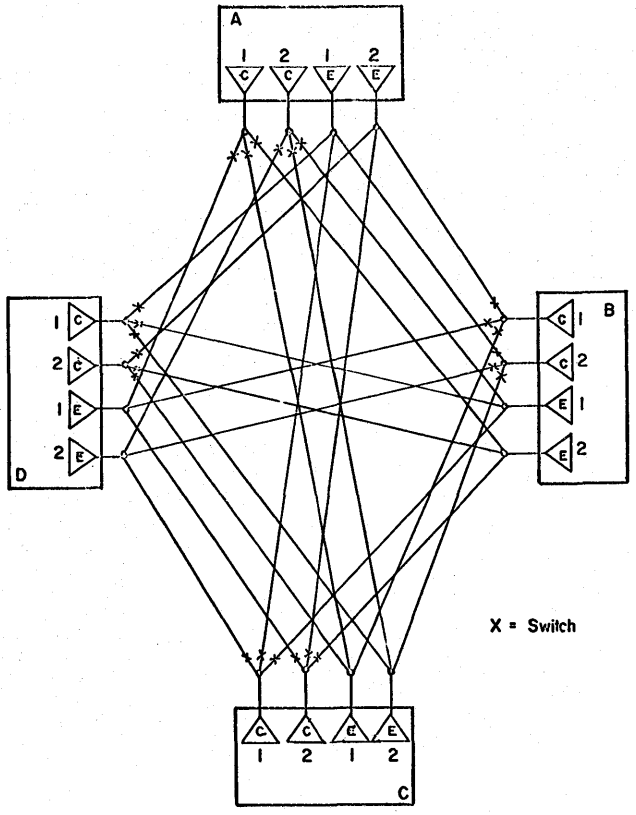


Figure 10—Small computer using nearly exhaustive interconnection routing subsystem

expander (input port) in each of the other modules. No two concentrators in a single module connect to the same expander.

Because of the symmetry (non-uniqueness) of the input/output ports and the ability of each output port to connect to one input port in every module, a simple heuristic routing algorithm can be used. Such an algorithm is proposed in Reference 7 and tested on the set of typical problems which formed the basis of the hardware design. The algorithm was successfully applied by hand to all the problems, taking no more than 1 minute/problem. For the particular model computer of the References 6,7, the decision to use this inter-module system instead of the cheapest one caused a 14% increase in the number of switches used in the complete system.

CONCLUSIONS

This paper presents a framework for considering the design of automatic patching systems for analog computers. The framework is generated by attempting to minimize the total systems costs and, as a result, includes control programming considerations as well as the switching matrix design. Experience gained from completing a design under this framework is discussed to illustrate the design procedure and indicate its effectiveness.

REFERENCES

1 M FRANKLIN J C STRAUSS
A hybrid computer programming system
Proceedings FJCC November 1969
2 C GREEN H D'HOOP A DEBROUX
APACHE—A breakthrough in analog computing
IRE Trans on Elec Computers October 1962
3 G HANNAUER
Stored program concept for analog computers
Final Report Parts 1 and 2 EAI Project #32009 Electronic Associates Inc Princeton New Jersey
4 G HANNAUER
Automatic patching for analog and hybrid computers
Simulation Volume 12 No 5 pp 219–232 May 1969
5 D STARR J JONNSSON
The design of an automatic patching system
Simulation June 1968
6 T GRACON J C STRAUSS
A decision procedure for selecting among proposed automatic analog computer patching system
Simulation September 1969
7 T GRACON
The systematic design of efficient automatic patching systems for analog computers
PhD Thesis Department of Electrical Engineering Carnegie–Mellon University Pittsburgh Pennsylvania 1969
8 K KUROKAWA
All IC hybrid computer eliminates the patchwork from programming
Electronics March 17 1969

18 bit digital to analog conversion

by J. RAAMOT

Western Electric Company, Inc.
Princeton, New Jersey

In many control applications it is necessary to convert the digital output from a computer to a proportional voltage to effect control. Incremental changes in this analog voltage become relatively smaller as the number of significant digits is increased, until a point is reached where it is not possible to distinguish between voltages caused by the smallest incremental number change.

It is common to have a computer output of 12, 16, or 18 significant binary digits (bits), or twice as many bits in double precision. There exists a problem in retaining the computer accuracy in the analog voltage. Commercially available digital to analog converters resolve voltage to 100 parts per million. This corresponds to less than 14 significant binary digits, even though the computer output may contain a higher accuracy.

A significantly better match in accuracy is achieved through the following construction: First, a 13 bit converter is used to represent the 13 least significant bits. Second, to the 13 bit converter are added high accuracy high order bits until they match the digital computer word length. This construction can be realized because of a new technique of obtaining accurate switchable voltage sources.

A digital to analog converter can be represented as shown in Figure 1. Each digit of a binary number appears at the computer output as one of two possible voltages that represent a logical 1 or a 0. The digital to analog converter uses a resistive voltage divider network to sum voltages from each digit in proportion to its significance. For example, digits of the binary number 1011 are summed as:

$$1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}.$$

Existing digital to analog converters use highly regulated voltage sources for the logical 1 and 0, and switch the resistors to the corresponding voltage source. Errors that are introduced by the switching element limit the accuracy to 100 parts per million.

The new technique is to use unregulated voltage sources, standard transistor switching elements, and to regulate the voltage after switching.

A standard technique for achieving a highly regulated voltage source is to use a series voltage regulating element. The new circuit, shown in Figure 2, uses just as effectively a parallel current source as the regulating element. It senses the voltage at the node (1) and introduces a current at the same point to keep the voltage at the same accuracy as a reference voltage, E_r.

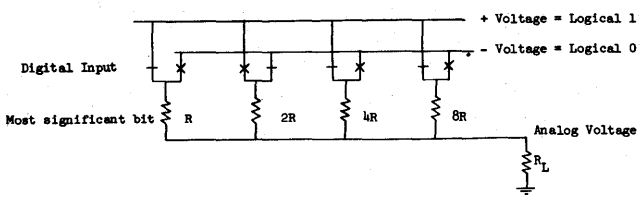


Figure 1—A 4 bit digital to analog converter with the binary input 1011

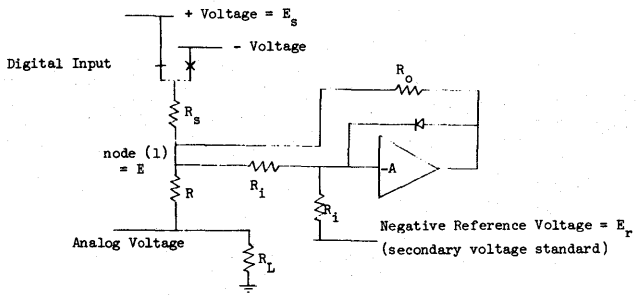


Figure 2—Amplifier circuit acts as parallel current regulator at node (1)

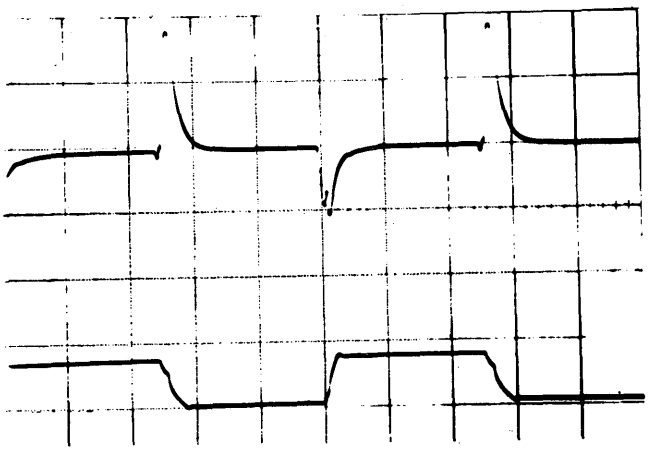


Figure 3—DAC switching waveforms between 377 777$_8$ and 400 000$_8$ at 5 us/cm and 10 v/cm. Top: DAC ouput—Bottom: Power source switching, voltage not to scale

In a noise-free analysis, the voltage at node (1) is:

$$E = -E_r + \frac{2}{A}\left(1 + \frac{R_0}{R_s}\right)E_r + \frac{2}{A}\left(\frac{R_0}{R_s}\right)E_s \qquad (1)$$

Equation 1 shows only the first order error terms. It is significant to recognize that the voltage E at node (1) is independent of the power source, E_s, and of the resistors R_0 and R_s, provided that the amplifier gain A, is sufficiently large.

A further analysis yields the result that a change in E due to a change in amplifier gain is proportional to A^{-2}. This suggests that if the gain is about 10^4 and the other terms are held constant to 1%, then it is possible to obtain a stability of 1 part per million.

The transient response of the circuit can be calculated by assuming the initial condition that all voltages are zero, and at zero time a step voltage of one volt is applied at the power source, E_s. Another assumption, that the amplifier frequency characteristic is equivalent to a low-pass RC network, results in the following transient voltage at node (1):

$$E(t) = \frac{R_0}{R_s}e^{-\pi F t} \qquad (2)$$

Again, only the first order terms are shown. In equation 2 F is the gain-bandwidth product of the amplifier.

The implicit time constant and settling time of the above model are in the nanosecond range. Unfortunately, the practical transient response is determined by propagation delay through the amplifier and by the slewing rate.

Likewise, the practical accuracy limitation is due to amplifier noise. A noise generator, E_n, representing the amplifier noise at its input will appear at node (1) as a voltage of $-2E_n$ superimposed on the voltage E.

Even with the above practical limitations, it is easy to realize the construction of an 18 bit digital to analog converter by the use of the regulation circuit of Figure 2.

Parallel current voltage regulators are limited in range by the current source. The circuit shown in Figure 2 is limited in range by the maximum available current from the amplifier. This property is put to good use in digital to analog conversion.

If the power source, E_s, is within regulation range, then the voltage, E, has the desired accuracy. If the power source is out of regulation range, then no regulation takes place. It is possible to switch the power source between two states, where for each state there is a regulation circuit that is within range. Thus, two regulation circuits are required for each high order bit of the digital to analog converter. This arrangement is driven by a reasonably fast power switching circuit.

It is necessary to design an 18 bit digital to analog converter for direct drive at the final voltage and current level. Contrary to common practice, there should be no amplifier in the converter output circuit. Any output amplifier would necessarily degrade the converter performance.

For the same reason it is difficult to evaluate the performance of an 18 bit converter. Figure 3 shows the transient response of an 18 bit converter at mid-range when all bits switch state. The least significant bit change is not resolved in this photograph. On the same

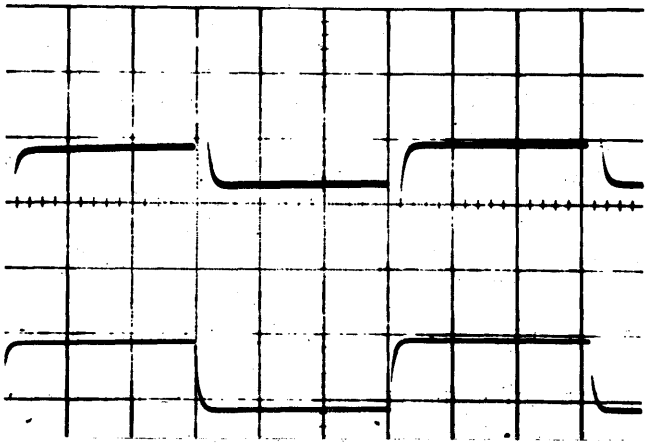


Figure 4—DAC switching waveforms between 377 777$_8$ and 400 000$_8$ at 500 us/cm and 1 mv/cm. Top: DAC output—Bottom: Power switching, voltage not to scale

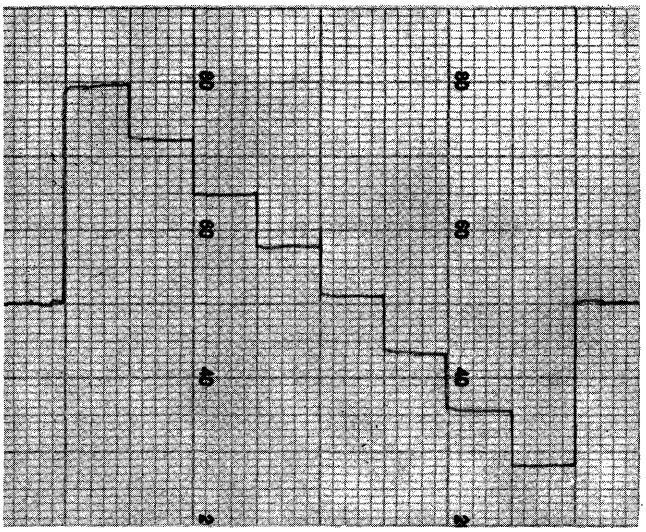


Figure 5—DAC ouput from 377774_8 to $400\ 003_8$ at 1 min/in and 1 mv/10 divisions

picture are shown the power source switching waveforms.

Figure 4 shows the same waveforms at a more sensitive oscilloscope range, but at less bandwidth. Here the least significant bit change is resolved. Figure 5 shows the dc resolution of least significant bit changes of the 18 bit converter.

The converter has an accuracy of 10 parts per million and has a stability of better than 1 part per million per day. This performance is obtained in an unprotected laboratory environment. Only the standard cell, that is used as the primary reference, is in a temperature controlled oven.

The stability and resolution obtained for the 18 bit converter suggest that the same converter could be extended to 20 bits. A true evaluation of the performance of high precision digital to analog converters can only be made in the final system where they are needed.

The technique of adding high order bits to existing digital to analog converters is very successful. It may appear to be expensive to construct the additional bits by having a separate regulation circuit for each state of each bit. However, this is the practical way to achieve high resolution.

A hybrid computer method for the analysis of time dependent river pollution problems

by R. VICHNEVETSKY

Electronic Associates, Inc. and Princeton University
Princeton, New Jersey

and

ALLAN W. TOMALESKY

Electronic Associates, Inc.
Princeton, New Jersey

INTRODUCTION

This paper is devoted to the description of work done in the hybrid computer simulation of polluted rivers and estuaries. Our attention in this paper is restricted to the solution of the pollutant concentration equation. The computer method used to perform the integration is essentially a continuous-space discrete-time method of lines. We have, in a previous paper,[1] described a continuous-space-discrete-time computer method for the analysis of flows and velocities in a one-dimensional river or estuary. Hence, these two programs, which may be exercised simultaneously, must be viewed as part of the same problem, since the pollutant diffusion parameters in a river (as described in the present paper) may be derived as explicit functions of the river geometry and water flow.

The kind of problems in partial differential equations to which river flows and pollution studies belong are as a rule computer time-consuming. It is therefore desirable to place emphasis on techniques by which truncation error-correction methods may lead to larger grid sizes in the finite differences processes of approximation. Such a truncation error characterization and correction method is embodied in the present paper, which permits the truncation error induced by larger time steps in the computer simulation to be (in the first approximation) corrected for in a semi-exact fashion.

PROBLEM STATEMENT

A simplified analysis of a one-dimensional river in terms of the polluting species is given mathematically as a partial differential equation representing the mass balance on the pollutant in one space dimension and time (see, e.g., Reference 6 for a derivation).

$$\frac{\partial c}{\partial t} = \frac{\partial}{\partial x} k \frac{\partial c}{\partial x} - \frac{\partial (V \cdot C)}{\partial x} - D(c) + f \qquad (1)$$

where:

$C = C(x, t)$ pollutant concentration
$V = V(x, t)$ river velocity (ft./sec.)
$D(c)$ = pollutant degradation or decay function (We shall assume for simplicity of the ensuing discussion that $D(c) = D \cdot C$ where D is a constant.)
$k = k(x, t)$ diffusion constant (ft²/sec.)
$f = f(x, t)$ pollutant source function
x = river length variable (ft.)
t = time variable (sec.)

The boundary conditions associated with this problem are discussed in a later section.

COMPUTER ANALYSIS

The CSDT approximation

The hybrid continuous-space-discrete-time method of approximation consists in expressing the solution along equi-distant lines parallel to the x axis in the (x, t) plane.
Call $c^j(x)$ the approximation of $c(x, t^j)$ where $t^j = j \cdot \Delta t$; $j = 0, 1, 2, \ldots$.
Then equation (1) may be approximated by the se-

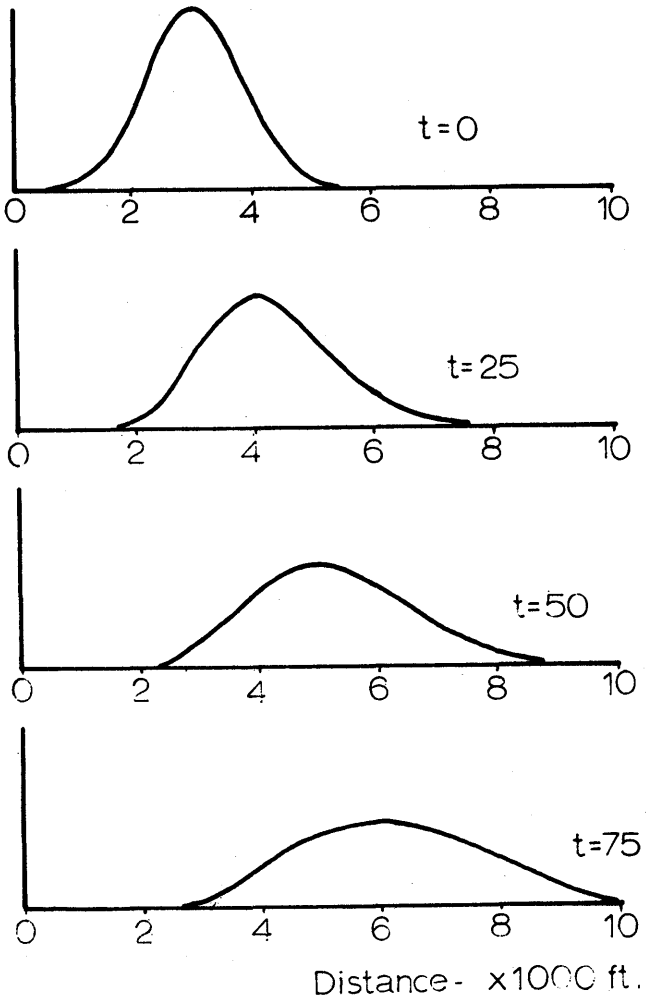


Figure 1—Typical propagation of pollutant profile

quence of ordinary differential equations (for $j = 0, 1, 2 \ldots$).

$$\frac{c^{j+1} - c^j}{\Delta t} = \theta\left[\frac{d}{dx}k^{j+1}\frac{dc^{j+1}}{dx} - \frac{d(V^{j+1}c^{j+1})}{dx} - Dc^{j+1} + f^{j+1}\right] + (1-\theta)\left[\frac{d}{dx}k^j\frac{dc^j}{dx} - \frac{d(V^jc^j)}{dx} - Dc^j + f^j\right] \quad (2)$$

where θ is a constant which must be chosen in the interval $\frac{1}{2} < \theta \leq 1$ to ensure stability in the time marching process.[2]

To produce a recurrence relation for the time marching solution of (2), we solve that equation for (c^{j+1}):

$$\frac{d}{dx}k^{j+1}\frac{d}{dx}c^{j+1} - V^{j+1}\frac{dc^{j+1}}{dx} - \left(\frac{1}{\theta\Delta t} + \frac{dV^{j+1}}{dx} + D\right)c^{j+1} = -\frac{c^j}{\theta\Delta t} - f^{j+1} - \left(\frac{1-\theta}{\theta}\right)\left[\frac{d}{dx}k^j\frac{d}{dx}c^j - \frac{d(V^jc^j)}{dx} - Dc^j + f^j\right] \quad (3)$$

Let the right-hand side equal:

$$-\frac{\bar{c}^j}{\theta\Delta t} - f^{j+1}$$

where:

$$\bar{c}^j = c^j + (1-\theta)\cdot\Delta t\left[\frac{d}{dx}k^j\frac{dc^j}{dx} - \frac{d(V^jc^j)}{dx} - Dc^j + f^j\right] \quad (4)$$

It is easily shown that $\bar{c}^j$ satisfies the recurrence relation:[3,4]

$$\bar{c}^{j+1} = c^{j+1} + \frac{(1-\theta)}{\theta}(c^{j+1} - \bar{c}^j) = \bar{c}^j + \frac{1}{\theta}(c^{j+1} - \bar{c}^j) \quad (5)$$

For convenience, we call R^j the entire right-hand side of equation (3):

$$R^j = -\frac{\bar{c}^j}{\theta\cdot\Delta t} - f^{j+1}$$

We can see that equation (3) can be rewritten as:

$$\frac{d}{dx}k^{j+1}\frac{dc^{j+1}}{dx} - V^{j+1}\frac{dc^{j+1}}{dx} - \left(\frac{1}{\theta\cdot\Delta t} + \frac{dV}{dx} + D\right)c^{j+1} = R^j \quad (6)$$

R^j is a known function of x and this equation can now be solved at each time step, together with the algebraic calculation of $\bar{c}^{j+1}$ as expressed by (5).

Stability problem and application of the method of decomposition

Equation (6) is of the second order in x. For constant V and k, its characteristic equation is:

$$k\gamma^2 - V\gamma - \left(\frac{1}{\theta\Delta t} + D\right) = 0 \quad (7)$$

or

$$\gamma = \frac{V}{2} \pm \sqrt{\frac{V^2}{4} + k\left(\frac{1}{\theta \cdot \Delta t} + D\right)} \tag{8}$$

These two values of γ are real and of opposite sign. Thus, direct integration of equation (6) as an initial value problem of the second order will have unstable error propagation properties which may impair the validity of the computer results.

The Method of Decomposition (Vichnevetsky,[2,3]) consists in avoiding the difficulty by transforming this second order differential equation into two first order ordinary differential equations, for which directions of stable x-integration may be chosen independently.

This is obtained as follows:

The second order operator

$$L = \frac{d}{dx} k \frac{d}{dx} - V \frac{d}{dx} - \left(\frac{1}{\theta \cdot \Delta t} + \frac{dV}{dx} + D\right) \tag{9}$$

appearing in equation (6) is (arbitrarily) decomposed into the product of two first order operators, L_B and L_F, which are intended to yield stable integrations in the backward and forward directions, respectively:*

$$\left.\begin{aligned} L &= L_B \cdot L_F \\ L_B &= \left(\frac{d}{dx} - \lambda_B(x)\right) \\ L_F &= \left(k(x)\frac{d}{dx} - \lambda_F(x)\right) \end{aligned}\right\} \tag{10}$$

Conditions for the stable integration in these respective directions are $\lambda_B \geq 0$ and $\lambda_F \leq 0$.

By identification of (10) with (9), we find:

$$\begin{aligned} L &= \frac{d}{dx} k \frac{d}{dx} - V \frac{d}{dx} - \left(\frac{1}{\theta \cdot \Delta t} + \frac{dV}{dx} + D\right) \\ &= L_B \cdot L_F \\ &= \frac{d}{dx} k \frac{d}{dx} - \frac{d\lambda_F}{dx} - \lambda_F \frac{d}{dx} - k\lambda_B \frac{d}{dx} + \lambda_F \lambda_B \end{aligned}$$

or:

$$k\lambda_B(x) + \lambda_F(x) = V(x) \tag{11}$$

* The operator $L_F(\,.\,)$ is said to be forward-stable if all solutions of the equation $L_F(v) = 0$ are stable in the classical sense. The operator $L_B(\,.\,)$ is said to be backward-stable if all solutions of the equation $L_B(v) = 0$ are stable in the classical sense when the integration variable $(-dx)$ is used instead of dx An operator $L(\,.\,)$ is said to be unstable when it is neither forward-stable, nor backward-stable.

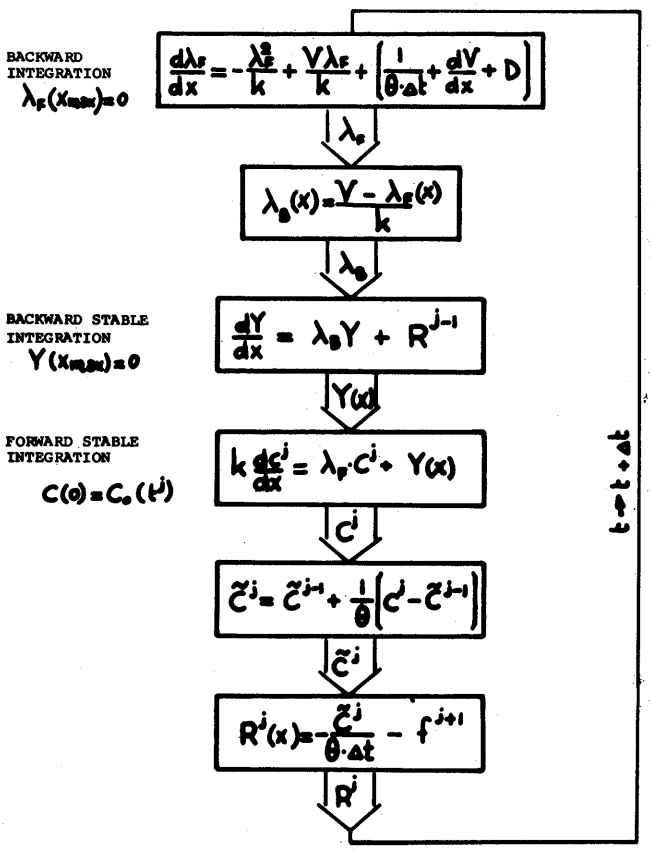


Figure 2—Computing sequence block diagram

and:

$$\frac{d\lambda_F}{dx} - \lambda_F\lambda_B = \frac{1}{\theta \cdot \Delta t} + \frac{dV}{dx} + D \tag{12}$$

λ_F may be obtained by the integration of the Ricatti equation:

$$\frac{d\lambda_F}{dx} - \lambda_F\left(\frac{V - \lambda_F}{k}\right) = \frac{1}{\theta \cdot \Delta t} + \frac{dV}{dx} + D \tag{13}$$

and λ_B subsequently obtained by the application of equation (11).

Now, a particular solution of equation (6) is obtained by the following sequence of computer integrations:

$$\text{(a):}\quad L_B(y(x)) \equiv \frac{d}{dx}y - \lambda_B(y) = R^j(x) \tag{14}$$

$$\text{(b):}\quad L_F(c(x)) \equiv k\frac{dc^{j+1}}{dx} - \lambda_F c^{j+1} = y(x) \tag{15}$$

Indeed, that c^{j+1} satisfies (6) is easily shown:

$$L(c^{j+1}) = L_B \cdot L_F(c^{j+1}) = L_B(L_F(c^{j+1})) = L_B(y) = R^j$$

q.e.d.

In summary, the sequence of equations solved at each time step in this problem is that of Figure 2.

BOUNDARY CONDITIONS

The equation (1) is of the first order in time and second order in space. Hence, solutions are specified by one initial condition function (i.e., the initial pollutant concentration profile $c(x, 0)$) and by two spatial boundary conditions. One of these boundary conditions, $c(0, t)$, is well defined (the pollutant concentration at the inlet of the river section under analysis), while the second boundary condition, $c(x_{max}, t)$ is not easily defined in terms of the problem formulation. However, that end-boundary condition has, mathematically, a very small influence on the solution $c(x, t)$ for $x \in [0, x_{max}]$ except for a small region which is close to x_{max}. Hence, one may choose this end-boundary condition to take any convenient form. The one chosen in the computer implementation described hereafter is that:*

$$\left.\frac{\partial c}{\partial x}\right|_{x_{max}} = 0 \tag{16}$$

This condition can be seen to be automatically satisfied by choosing as boundary conditions for the λ_F and y equations (equations (13) and (14), respectively):

$$\left.\begin{aligned} \lambda_F(x_{max}) &= 0; \\ y(x_{max}) &= 0 \end{aligned}\right\} \tag{17}$$

i.e., from equation (15):

$$\left.\frac{dc^{j+1}}{dx}\right|_{x_{max}} = \frac{1}{k}\left[y(x_{max}) - \lambda_F(x_{max}) \cdot c^{j+1}\right] = 0 \quad \text{q.e.d.}$$

ERROR ANALYSIS AND FIRST ORDER CORRECTION OF THE CSDT APPROXIMATION

Analysis

Application of the CSDT approximation to the simple fluid transport equation:

$$\frac{\partial u}{\partial t} = -V\frac{\partial u}{\partial x} \tag{18}$$

introduces a truncation error which has the effect to "disperse" the solution $u(x, t)$ by a diffusion-like phenomenon. Hence, one may look at the CSDT approximation of equation (1) as an approximation process in which the diffusion coefficient $k(x, t)$ results from that which is introduced explicitly by the computing process described in an earlier section of this paper, plus a spurious $k^*(x, t)$ which is introduced by the CSDT approximation itself. If the spurious part of the diffusion coefficients (i.e., $k^*(x, t)$) can be predicted, then it becomes an easy matter to correct for this factor by subtracting it from the desired value before entering into the computing sequence of the third section of this paper.

The remainder of this section is an analysis of the truncation-induced diffusion effect of the CSDT approximation of equation (18) followed by an experimental computer verification of the applicability of these theoretical results to the more general CSDT approximation of the transport-diffusion equation (1), as described earlier. The partial differential equation (18) describes a pure fluid transport phenomenon.

The CSDT approximation of equation (18) is expressed by:

$$\frac{u^{j+1} - u^j}{\Delta t} = -V\left[\theta\frac{du^{j+1}}{dx} + (1-\theta)\frac{du^j}{dx}\right] \tag{19}$$

The solution of (19) approximates that of a transport diffusion equation of the form:

$$\frac{\partial u}{\partial t} = -V\frac{\partial u}{\partial x} + k^*\frac{\partial^2 u}{\partial x^2} \tag{20}$$

where the diffusion constant k^* is a spurious diffusion coefficient, introduced strictly by the approximation process, and which depends on the parameters appearing in (19).

An equivalent value of k^* may be estimated analytically. To that effect, we express the different terms of (19) in a Taylor Series around the point $u^j(x)$:

$$u^{j+1} = u^j + \frac{\partial u}{\partial t}\cdot\Delta t + \frac{\partial^2 u}{\partial t^2}\cdot\frac{\Delta t^2}{2} + \cdots \tag{21}$$

$$\frac{du^{j+1}}{dx} = \frac{\partial u^{j+1}}{\partial x} = \frac{\partial u^j}{\partial x} + \frac{\partial^2 u^j}{\partial x \partial t}\cdot\Delta t + \cdots \tag{22}$$

Hence, upon substitution of these relations in (19), that equation becomes (we may now delete the superscripts):

$$\frac{\partial u}{\partial t} + \frac{\Delta t}{2}\frac{\partial^2 u}{\partial t^2} + \cdots = -V\left[\theta\left(\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x \partial t}\cdot\Delta t + \cdots\right) + (1-\theta)\cdot\frac{\partial u}{\partial x}\right] \tag{23}$$

* This assumption is not a limitation of the method described in this paper. Any other boundary condition $C(x_{max}, t)$ could be chosen. This then would require the independent calculation of solutions of the homogeneous equation $L(w) = 0$. as shown for instance in Reference 4.

For the exact solution, we have the relation:

$$\frac{\partial u}{\partial t} = -V\frac{\partial u}{\partial x}$$

which yields:

$$\frac{\partial^2 u}{\partial x \partial t} = -V\frac{\partial^2 u}{\partial x^2} \tag{24}$$

and

$$\frac{\partial^2 u}{\partial t^2} = +V^2\frac{\partial^2 u}{\partial x^2} \tag{25}$$

Thus, after using these relations, (23) becomes:

$$\frac{\partial u}{\partial t} = -V\frac{\partial u}{\partial x} + (\theta - \tfrac{1}{2}) \cdot V^2 \cdot \Delta t \cdot \frac{\partial^2 u}{\partial x^2} + \cdots \tag{26}$$

By identification of (26) with (20), we find the equivalent diffusion constant:

$$k^* = (\theta - \tfrac{1}{2}) V^2 \cdot \Delta t \tag{27}$$

For $\theta < \frac{1}{2}$, k^* becomes negative: It is of interest to note that this corresponds exactly to the values of θ for which the CSDT approximation is unstable.[2]

Computer verification of the analysis

Computer verification of the preceding analysis has confirmed its first-order validity within a range of parameters which applies to river pollution problems. In order to perform this verification, the homogeneous equation:

$$\frac{\partial c}{\partial t} = -V\frac{\partial c}{\partial x} + k_c\frac{\partial^2 c}{\partial x^2} \tag{28}$$

was integrated in the manner described in an earlier section on the computer, with initial conditions $c(x, 0)$ corresponding to a Gaussian distribution; i.e.,

$$c(x, 0) = \frac{A}{\sigma(0)} \cdot \exp - \frac{(x - x_p)^2}{2\sigma(0)^2} \tag{29}$$

where A is a positive constant, x_p the point where the peak of the distribution occurs at $t = 0$, and $\sigma(0)$ the initial standard deviation of the $c(x, 0)$ distribution.

The exact solution of (28) with (29) as initial condition is (at least if the boundaries are assumed to be far enough not to have any effect upon the solution):

$$c(x, t) = \frac{A}{\sigma(t)} \cdot \exp - \frac{[x - (x_p + V \cdot t)]^2}{2 \cdot \sigma(t)^2} \tag{30}$$

where $\sigma(t)$ is the solution of:

$$\frac{d\sigma}{dt} = \frac{k}{\sigma}$$

Hence,

$$\sigma(t)^2 - \sigma(0)^2 = 2k \cdot t \tag{31}$$

and, generally, between two instants of time t_1 and $t_2 > t_1$:

$$\sigma(t_2)^2 - \sigma(t_1)^2 = 2k(t_2 - t_1) \tag{32}$$

Equation (30) expresses the fact that the "peak" moves with the flow at the velocity V, that the Gaussian distribution property remains preserved in time, and that the standard deviation $\sigma(t)$ grows as $\sqrt{t}$.

Experimental measurement of $\sigma(t)$ is easily achieved, either by measuring the "peak" of the solution:

$$c_{\max}(t) = \frac{A}{\sigma(t)} \tag{33}$$

or by measuring the "2σ" of $c(x, t)$ at $1/\sqrt{e}$ of the peak: (for $c = c_{\max} \cdot e^{-1/2} \cdot x = x_{\text{peak}} \pm \sigma$). A typical input is shown in Figure 1.

For the purpose of this study, the computer program described in an earlier section was utilized for equation (28), where k_c was chosen "small" (specifically equal to .01 ft/sec^2), and the results shown on Figure 3 are,

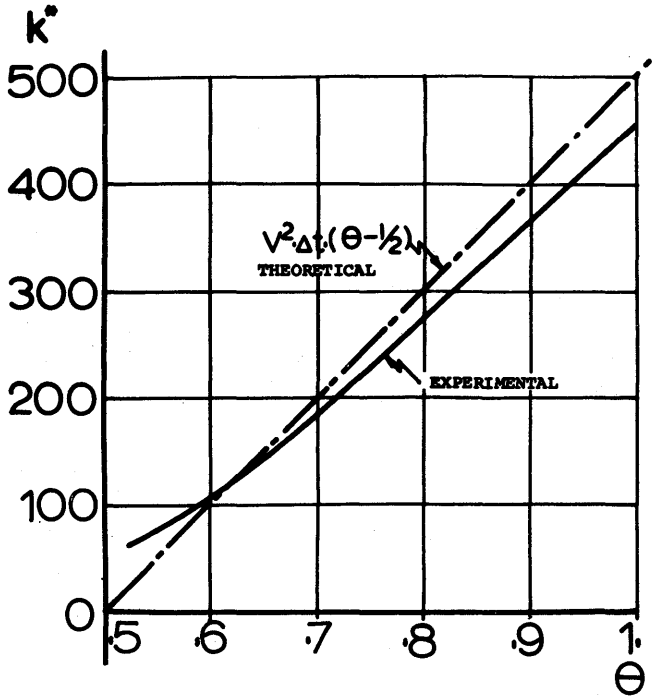


Figure 3—K^* vs. θ for $V = 4.$ and $\Delta t = 50$

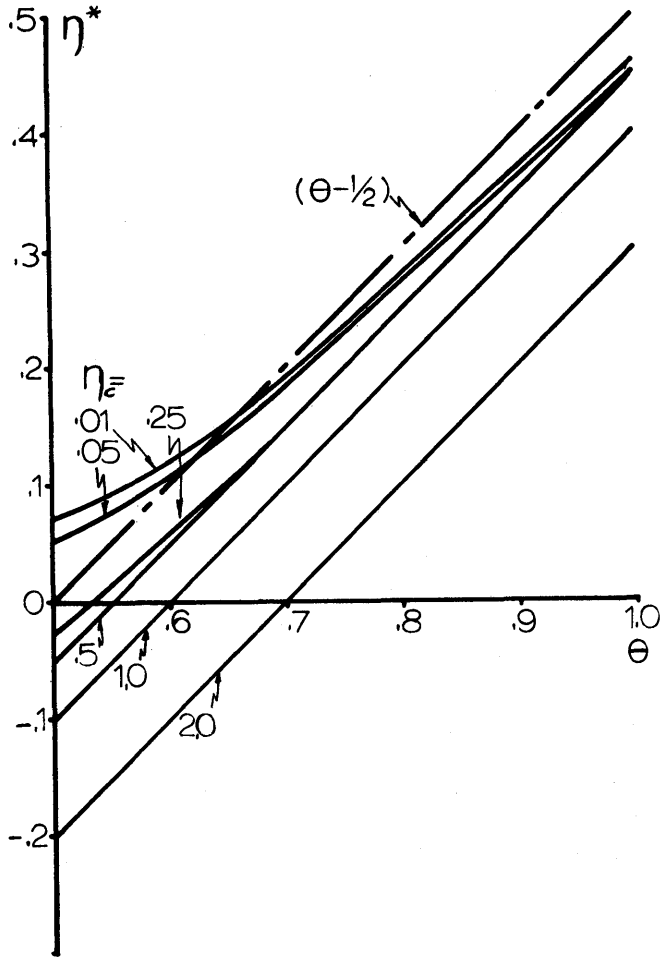


Figure 4—η^* vs. θ

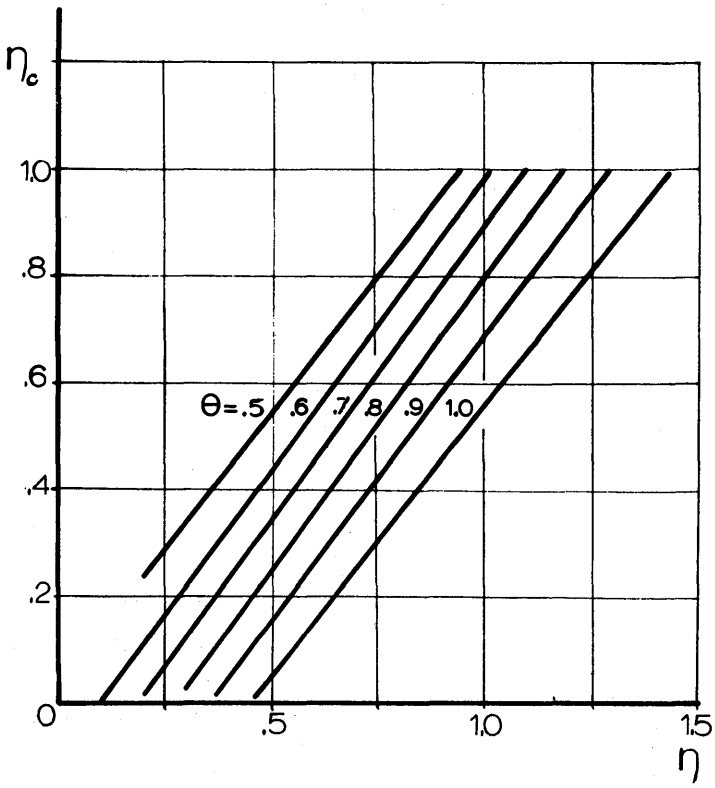


Figure 5—η_c as a function of η

in effect:

$$k^* = k_{\text{measured}} - k_c \simeq k_{\text{measured}} \quad (k_c \text{ small})$$

(Note that the range of k of interest for river studies is of 500 and over.)

Experimental results are shown in Figure 3. Consideration of this figure shows that there is a reasonably good agreement between the predicted and experimental values of the truncation-induced diffusion constant k^*.

Induced diffusion in the transport-diffusion equation

The preceding analysis and computer verification of k^* is concerned with the simplified transport equation

$$\frac{\partial c}{\partial t} = -V\frac{\partial c}{\partial x},$$

or at least with the transport diffusion equation with "small" values of the diffusion constant k.

We are in practice interested in the equations of the form:

$$\frac{\partial c}{\partial t} = -V\frac{\partial c}{\partial x} + k\frac{\partial^2 c}{\partial x^2} + g$$

where k is not "small" in the sense previously described.

The question thus arises as to what happens to k^* as a function of θ, for non-small values of k. An answer to this question was searched experimentally, by performing experiments similar to that described in the preceding section, but with k as an additional free parameter. In this analysis, it was first recognized that an analysis of the *dimensionless* "induced" diffusion constant $\eta^* = k^*/V^2 \cdot \Delta t$ could be obtained as a function of the *dimensionless* "explicit" diffusion constant $\eta_c = k_c/V^2 \cdot \Delta t$, thereby providing a relationship where, for a fixed value of θ, η^* would be a function of η_c alone.

The argument here is that there is no reason why Buckingham's π principle cannot be applied to the behavior of computer program solutions as it applied to the field of mechanics and thermodynamics. Hence, relationships between dimensionless parameters must be absolute, save for round-off phenomena, if and where they occur.)

Experience confirmed this theory, and Figure 4 shows a dimensionless chart of the induced $\eta^* = k^*/V^2 \cdot \Delta t$ as a function of θ and $\eta = k/V^2 \cdot \Delta t$.

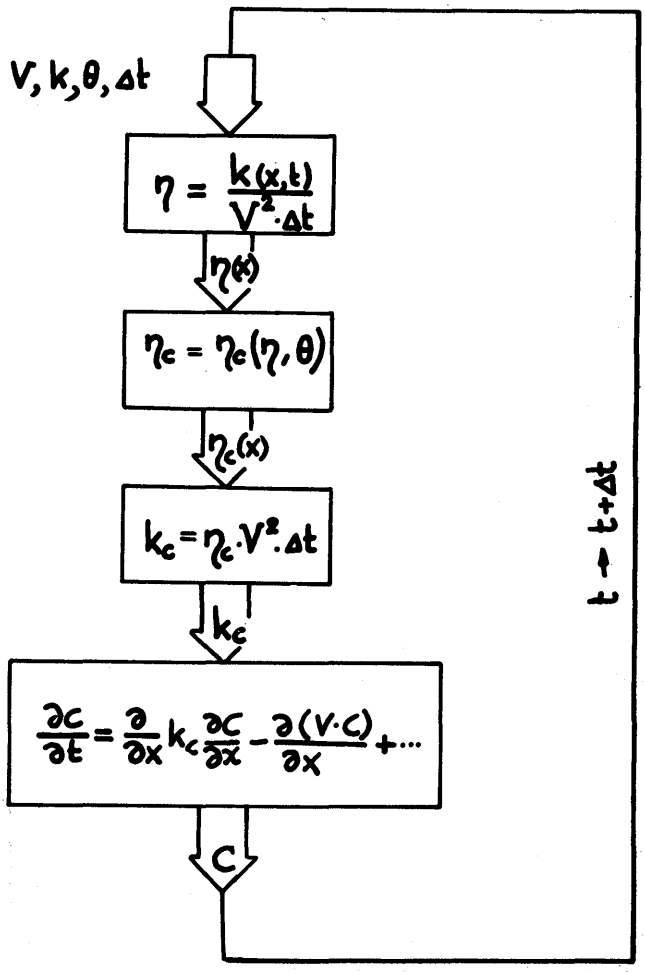


Figure 6—Diffusion-corrected computer method

We are reminded at this point that any practical computer program will entail a fixed value for θ, and that η^* will thereby become a function of η alone.

Figure 5 shows this more useful relationship for various values of θ.

DIFFUSION-CORRECTED COMPUTER METHOD

The "diffusion-corrected" method simply consists in deriving the function $\eta_c(\eta)$ for the particular value of θ being used from Figure 5 and introducing the corrected diffusion constant $k_c = \eta_c V^2 \Delta t$ into the procedure discussed earlier.

This correcting method is applied continuously as a function of time and space, as shown in Figure 6.

CONCLUSIONS

The method of simulation of river pollution described in this paper has been implemented as a computer program. Experimental results have confirmed the usefulness of the diffusion correction of Section 6 as a means of allowing larger time steps to be utilized. It has also been found that the general "hybrid approach" which consists in approximating the problem in the form of ordinary differential equations offers a convenient way to implement pure-numerical simulations.

REFERENCES

1 R VICHNEVETSKY
Computer integration of hyperbolic partial differential equations by a method of lines
Proc Fourth Australian Computer Conference Adelaide South Australia August 1969

2 R VICHNEVETSKY
Hybrid computer methods for partial differential equations
Course Notes for the In-Service Seminar on Engineering Applications of Hybrid Computation Pennsylvania State University June 19–21 1969

3 R VICHNEVETSKY
Application of hybrid computers to the integration of partial differential equations of the first and second order
Proceedings of the IFIP Congress 68 Edinburgh Scotland August 5–10 1968

4 R VICHNEVETSKY
A new stable computing method for the serial hybrid computer integration of partial differential equations
Proceedings SJCC Conference Vol 32 Thompson Book Company Washington D C 1968

5 LANGHAAR
Dimensional analysis and theory of models
J Wiley and Sons New York New York 1951

6 R BIRD W STEWART E LIGHTFOOT
Transport phenomena
J Wiley and Sons New York New York 1960

Programmable indexing networks

by KENNETH JAMES THURBER

Honeywell Incorporated
St. Paul, Minnesota

INTRODUCTION

One of the most important functions that must be performed in a digital machine is the handling and routing of data. This may be done in routing logic (computers), in permutation switching networks (computers and telephone traffic), sorting networks, etc. In some parallel processing computers being envisioned the handling of large blocks of data in a parallel fashion is a very important function that must be performed. For a special-purpose machine a fixed-wire permutation network could be acceptable for the handling of data; however, for a general-purpose machine more sophisticated reprogrammable networks are required.

The permutation network problem has been previously studied by Benes,[2] Kautz et al.,[3] Waksman,[4] Thurber,[5] and Batcher.[1] This paper introduces and defines a new network to be considered. This is the *generalized indexing network*. This network can perform an arbitrary mapping function and is easily reprogrammable to perform any other arbitrary map with n inputs and m outputs, and has many potential areas of use. The most interesting possible area of application is the processing of data while routing the data. If the network is used as routing logic, it can perform many simple data manipulation routines while routing the data e.g., matrix transposition.

Some of the solutions presented are significant improvements on the shift register permuters suggested by Mukhopadhyay.[7] The solutions suggested here are programmable (utilizing the output position mask), as fast, and utilize less hardware than the previously suggested shift registers permuters.

FORMULATION OF THE PROBLEM

Previously, most researchers have considered the problem of permuting a set of n input lines $X_1, X_2, \cdots, X_{n-1}, X_n$ onto a set of n output lines $Y_1, Y_2, \cdots, Y_{n-1}, Y_n$ by means of a device called a permuter. A permuter produces a one to one mapping from the n input lines to the n output lines of the network. The permuter can perform a very limited set of functions. As currently studied, the permutation networks can only transfer lines of data. In this paper the networks will be utilized to transfer words of data.

Limitations of permutation networks are that input words cannot be repeated or deleted at the output. Also, blanks cannot be inserted into the output and the number of input words and the number of output words must be equal. The *indexing network** differs from the permuter in that input words can be repeated or deleted and blanks can be inserted in the output. Also, for an indexing network the number of input words (n) has no special relation to the number of output words (m). The non-blank output words may appear in many contiguous subsets of the output words (these subsets could be empty). Figure 1 shows some examples of possible permutation networks. Figure 2 shows some examples of possible indexing networks.

In this paper X_i means a word of input data (instead

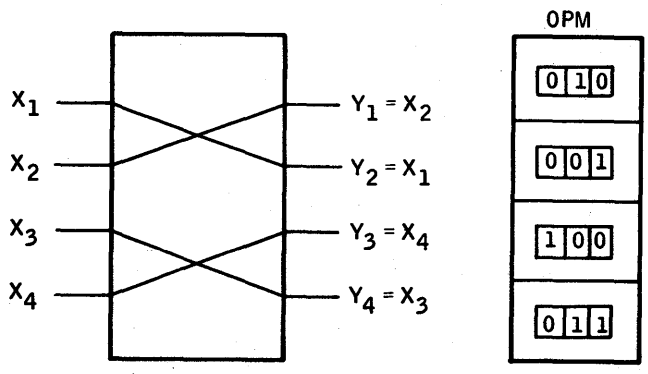


Figure 1—Permuter

* The terminology *indexing network* and *generalized indexing network* will be taken to have the same meaning.

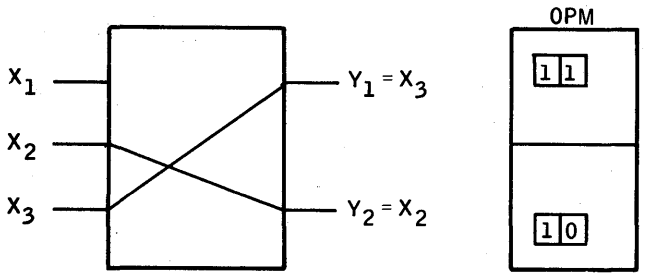


Figure 2—Indexing network and its OPM

of an input line) and Y_i means a word of output data (instead of an output line). The blank word is designated by 0. The actual storage device containing or receiving the word of information (or the number of bits in the word) is not shown and the inputs and outputs to the network are still only pictured as one line for each X_i and Y_i. (When in reality each line may symbolically represent p parallel input lines (for a p bit word) for the parallel transfer of each word into and out of the network.) Each storage device for one word of information is called a *cell.*

It should be noted that the permutation network problem is a sub-problem of the generalized indexing network problem.

If N is a network with n inputs and m outputs then the *output position mask* (OPM) is a vector containing m distinct cells with $\log_2 (n + 1)$ binary bits per cell.* Each cell contains the binary code corresponding to the input value desired in the corresponding output cell. $\text{Log}_2 (n + 1)$ bits are needed since the n inputs and the 0 must have a code so that they can be specified as output values if desired. Figures 1 and 2 show several networks, along with their corresponding output position masks. Each cell consists of a shift register capable of delivering its contents (in parallel) onto the appropriate control lines of the network.

A SHIFT REGISTER SOLUTION

This is the first of several "shift register solutions" to be presented in this paper. The name shift register solution has been used for simplicity; however, what is actually used is a set of shift registers (each contains or receives one word of information) which can perform a parallel transfer of its contents to its neighbor. The transfers are arranged such that a transfer pulse to the input set of registers causes the simultaneous parallel cyclic transfer of the contents of the registers; i.e., $n \rightarrow n - 1, n - 1 \rightarrow n - 2, \ldots, 2 \rightarrow 1, 1 - 0$, and $0 - n$ simultaneously. A transfer pulse to the output set of registers (and to the OPM) causes the simultaneous parallel transfer of the contents of the registers; i..e, $n \rightarrow n - 1$ $(\text{OPM}(n) \rightarrow \text{OPM}(n - 1)), \ldots, 2 \rightarrow 1$ $(\text{OPM}(2) - \text{OPM}(1))$, and $1 \rightarrow n$ $(\text{OPM}(1) \rightarrow \text{OPM}(n))$. The previously specified functions are performed by the Input Cyclic Control (ICC) and Output Cyclic Control (OCC) respectively. The Transfer Control (TC) performs the function of transferring data from input position 0 to output position 1. There is no output position 0.

Figure 3 shows the clocking hardware used to read the OPM and produce the desired control pulse for the TC. It is assumed that the clocking hardware contains a clock with clock rate c/p, where c is the clock rate of the sorter and p is a suitable positive integer. Binary constants c_2, c_1, and c_0 placed on the input lines to the network produce an output from the network after $(c_2(4) + c_1(2) + c_0(1))$ units of delay. One unit of delay is equal to the time period between indexing clock pulses (the clock rate of the indexing network is c so a unit delay is c^{-1} second). The clocking hardware is used to advance the input registers to a position selected by the OPM.

Figure 4 shows a general setup for an indexing network and a complete indexing network for $n = 5$ and $m = 4$. The words are 4-bit words in this example. The indexing network consists of an input set of registers and associated ICC and TC hardware, an output set of registers and the associated OCC and OPM hardware, and the clocking and control hardware.

The clock rate of the indexing network is c per second

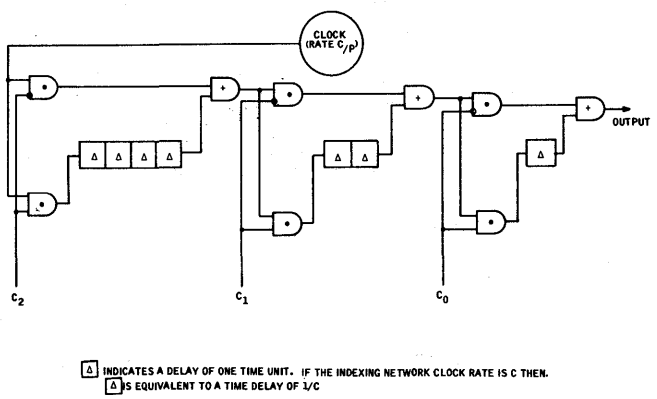


Figure 3—Clocking hardware for obtaining delays from 0 to 7 time units of delay

* Where $\log_2(n + 1)$ is understood to be rounded off to the next larger integer if $\log_2(n + 1)$ is not an integer; e.g., $\log_2(7 + 1) = 3$ and $\log_2(10 + 1) = 4$.

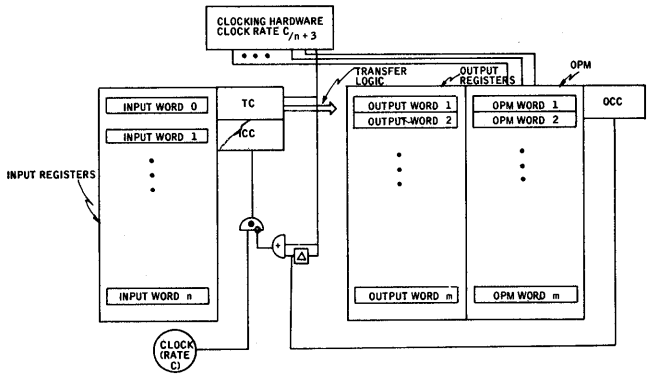


Figure 4(a)—Generalized shift register indexing network

and the clock rate of the clocking hardware is $c/8$ per second. In general the clock rate for the clocking hardware is $c/n + 3$ per second.* No provisions have been shown for connecting the network to other hardware, but this should be obvious. A blank (binary 0) is placed in register 0 of the set of input registers.

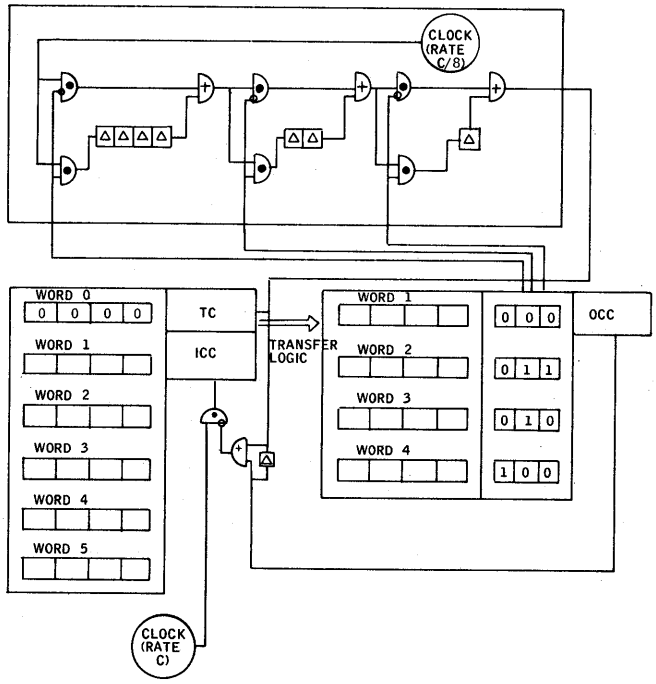


Figure 4(b)—Indexing network with $n = 5$, $m = 4$, and word lengths of 4 bits with the OPM set to produce $(0\ X_3\ X_2\ X_4)$

* $c/n + 3$ are needed instead of c/n because (1) a time period is needed for shifting $n + 1$ input values instead of just n input values, (2) a time period is needed to transfer the data, and (3) a time period is needed to shift the output registers and OPM.

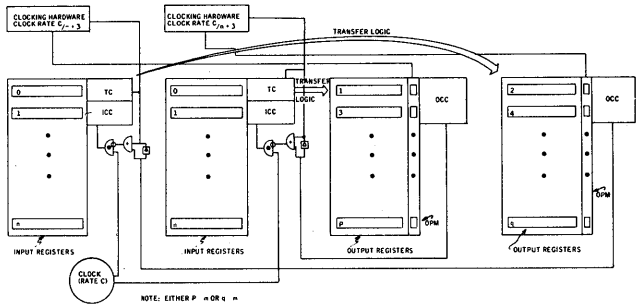


Figure 5—Indexing network

The operation of the network is easily explained. Assume the input registers are full and the first clock pulse is produced (in both the clocking hardware clock and the indexing network simultaneously). The binary value in the OPM causes the pulse to the ICC and TC to be delayed a number of time periods equal to its value. Meanwhile the input is being cycled. When the correct input register has moved into position 0, the transfer pulse arrives inhibiting further cycling and causing the transfer (a non-destruct read) from input 0 to output 1 to occur. The input is still inhibited and the output is shifted one position by the OCC. The input register then is cycled to its original state and the process begins again. After m cycles the output registers are all filled and back in their correct position so that the indexing operation has been completed.

This type of an indexing network can be configured in many different ways depending upon the speed desired and the hardware available. Figure 5 shows the manner in which the network could be set up for faster operation. The network in Figure 5 requires twice as much hardware as the network in Figure 4, but is twice as fast. Figure 6 is an indexing network that operates approximately n times as fast as the

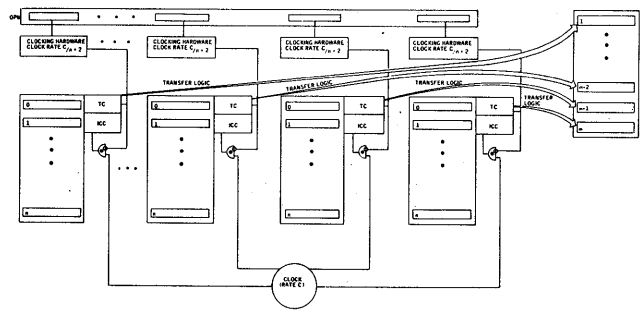


Figure 6—High-speed indexing network

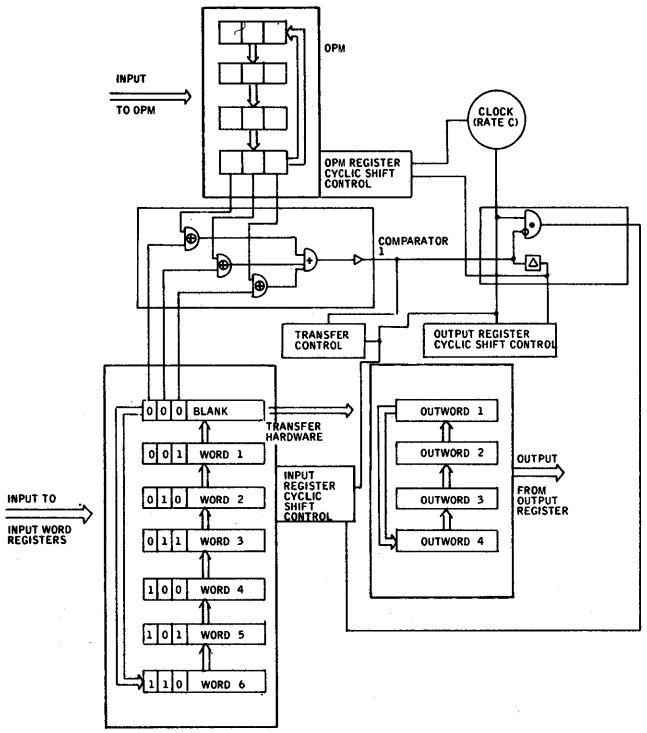


Figure 7—Comparator indexing network for $n = 6$ and $m = 4$

network in Figure 4. As can be easily seen, this solution to the generalized indexing network problem can be easily configured to account for many different hardware and speed requirements. In Figure 6, less logic is required in parts of the network and the clock rate of the clocking hardware is different than the rate of the network in Figure 4. This is because the set of output registers does not have to be shifted to their next receiving positions since the network is a "parallel" indexing network and the output is available after $n + 2/c$ seconds.

A SOLUTION UTILIZING SIMPLIFIED CLOCKING HARDWARE

The purpose of this section is to introduce another version of a generalized indexing network which utilizes shift registers to perform the indexing operation. This solution utilizes the OPM to program the network. Figure 7 shows the solution for $n = 6$, $m = 4$. An extra set of $\log_2 (n + 1)$ bits has been added to the input register. These bits contain the input position of the input data and are utilized to select the appropriate output value.

The details of the operation are as follows:

(1) The input data and the OPM are inputted into the network.
(2) The input data is cycled until the input code equals the current value of the OPM.
(3) The input word is transferred to the output register.
(4) The output register and the OPM are advanced one position unless the output register is full in which case go to (6).
(5) Go to (2).
(6) Output the data in the output register.
(7) Stop.

EXTENSIONS OF THE SOLUTION GIVEN IN PREVIOUS SECTION

The solution given in the previous section is interesting in that there are several other methods by which it can be implemented in a more sophisticated manner. Since the solution given previously does not require as much hardware as some of the other solutions it is interesting to consider what can be done with the addition of some extra hardware.

As with the solution given in the third part of this paper, the solution given in the previous section can be implemented in a form such as in Figures 5 and 6. Also, it could be implemented in any form that "lies" between the solutions given in Figures 5 and 6.

The following solutions require that the set of input registers be able to shift cyclicly backwards ($0 \rightarrow 1$, $1 \rightarrow 2, \ldots, n - 1 \rightarrow n, n \rightarrow 0$) as well as forwards ($1 \rightarrow 0, 2 \rightarrow 1, \ldots, n \rightarrow n - 1, 0 \rightarrow n$).

One method of improving the solution given previously is to make more than just a comparison of the two numbers for equality. A solution is to check and see whether the number contained in the OPM is greater than, equal to, or less than the number designating the current state of the input. If the OPM number is larger shift the input register forward, if the OPM number is smaller shift the input register backwards, and if the numbers are equal then transfer the information. The actual shifting can be implemented as in the previous section (a comparison after every input shift) or as in the third section (this would require a subtraction to determine the number of needed periods of delay) using the clocking hardware in Figure 3 to produce the transfer pulse.

Another improvement that can be made is based upon the following observation; i.e., if the set of registers can cycle both forwards and backwards then there are cases where it is shorter time wise to go around one of

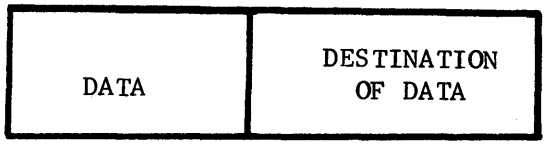


Figure 8—General arrangement of a splitter register

the "ends" of the set of input registers. For example, if $n = 10$ and the network is at 9 and needs to go to 1 then the shortest way is $9 \rightarrow 10$, $10 \rightarrow 1$ (instead of $9 \rightarrow 8$, $8 \rightarrow 7$, . . . , $2 \rightarrow 1$). This solution can be implemented by calculating and comparing $n + 1 - |p - q|$ to $|p - q|$ were p and q are the current location and the desired location. Again this solution could be built as in the previous section (comparison after each input shift) or as in the third section (using clocking delays); however, it is probably best implemented using clocking hardware (such as in Figure 3) because the minimum of $n + 1 - |p - q|$ and $|p - q|$ give the number of time delays to be produced by the clock. Therefore, after the comparison has been made, the minimum value can be used as input data into clocking hardware and the register cycled in the proper direction (forward or backward).

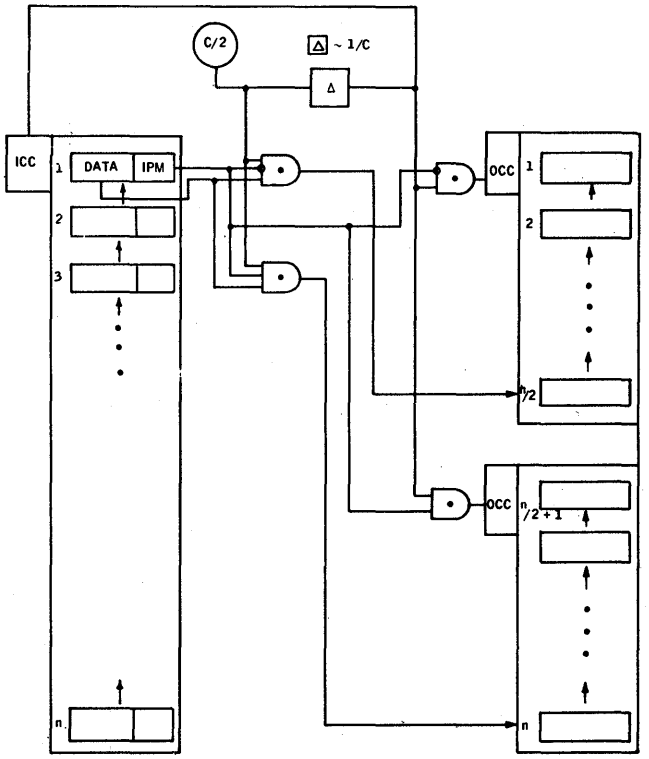


Figure 10—Section of the splitter used to produce a permutation

THE SPLITTER

This section presents a solution to the generalized data indexing problem based upon an input decision called the *input position map*. This solution utilizes a modular construction and seems most interesting in the case in which a lot of different indexings must be produced in rapid succession. A major advantage of this type of network is that it is capable of simultaneously processing many indexings at the same time.

The *input position map* (IPM) is a set of binary codes associated with the input data of a network that specifies the position (or positions) that the data is to be transferred to in the set of output registers. In the case of the design of a splitter it will be assumed that the input data and the binary code contained in the IPM associated with that input data are contained in an extended register as shown in Figure 8.

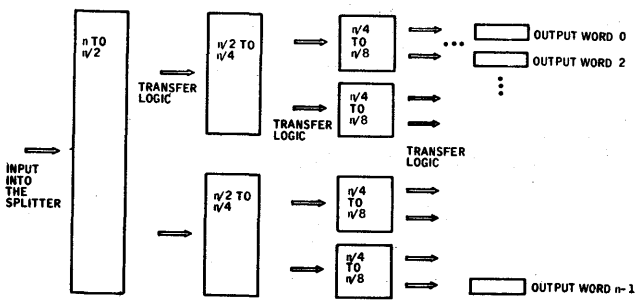


Figure 9—Use of splitters to perform a permutation for $n = 2^k$

Figure 11 shows the general block diagram of several splitter networks organized to perform a permutation function. Each module in the splitter takes the n inputs (assume n is even) and groups of these n inputs into two $n/2$ input groups based upon the mapping information contained in the mapping information portion of the node. The splitter is most useful in constructing sorting networks that have $n = 2^k$.

The permutation network shown in Figure 10 can be built in various sizes so that it can be configured as shown in Figure 9. The mapping information inputted to this network would be the binary value of the position in the set of output registers that the data was destined for so that an arbitrary input register would contain DATA and DESTINATION OF DATA where the destination of the data is between 0 and $n - 1$. The first splitter encountered ($n \rightarrow n/2$) would sort the information based upon the binary value contained in the highest order digit; whereas, the last group of

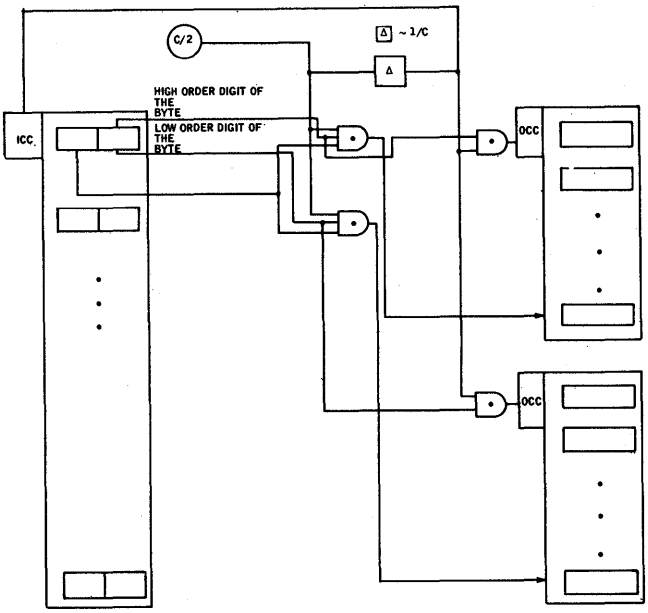


Figure 11—General splitter module

splitters ($2 \rightarrow 1$) would read the lowest order digit. The values being read would be inputted to the AND gates as shown in Figure 10. The full word of data would be transferred to the appropriate output register in parallel and the appropriate output register and the input register would be advanced one position each. The next word is then processed in the identical manner. To split n elements into two $n/2$ element groups requires n clock periods. The bit that the AND gate reads is different at each level, but begins with the high order digit and proceeds to the low order digit.

The IPM for a permuter constructed by the splitter method is just the binary output destinations of the data. It is a little harder to construct a generalized indexing network using this concept. The permuter was easy because it needed a one to one and onto mapping function. A generalized indexing network is a little harder but not impossible. It will be slightly harder to compute the IPM than it was for the permuter, but the following method and the hardware shown in Figure 11 configured as in Figure 9 will produce a generalized indexing network. One modification of the network is that in the first splitter, the data must be broken from n into two groups of $m/2$ elements. From that point on each group of $m/2^p$ elements is split into two groups of $m/2^{p+1}$ elements. The mapping information for the network can be furnished by the following observations. Each element of input data can be categorized as to where it is transferred by means of a two-bit binary map (byte). The high order byte specifies the split $n \rightarrow m/2$; whereas, the low order byte specifies the split $2 \rightarrow 1$. There are exactly four distinct possibilities that can happen to a piece of data; i.e., the data not transferred to either output register, the data transferred to one but not the other output, register (two possible cases), or the data transferred to both output registers. These are indicated in Figure 12 and the necessary hardware shown in Figure 11. This design allows design of a generalized indexing network if the output registers are all set to the blank (0) value before they receive any data. In order to make the splitter work utilizing two bit bytes, the mapping information must be introduced at each stage of the process as shown in Figure 13. If the mapping information was completely specified with the data in stage 1 there would be no way to produce the indexing (X_4 00 X_4) because the second byte would have to be 10 and 01 simultaneously. (X_4 00 X_4) could be produced by the map 11 associated with X_4 at stage 1 the map 10 associated with the value of X_4 in stage 2 (A^1), and the map 01 associated with X_4 a stage 2 (B^1) in Figure 13. The difficulty encountered in constructing the

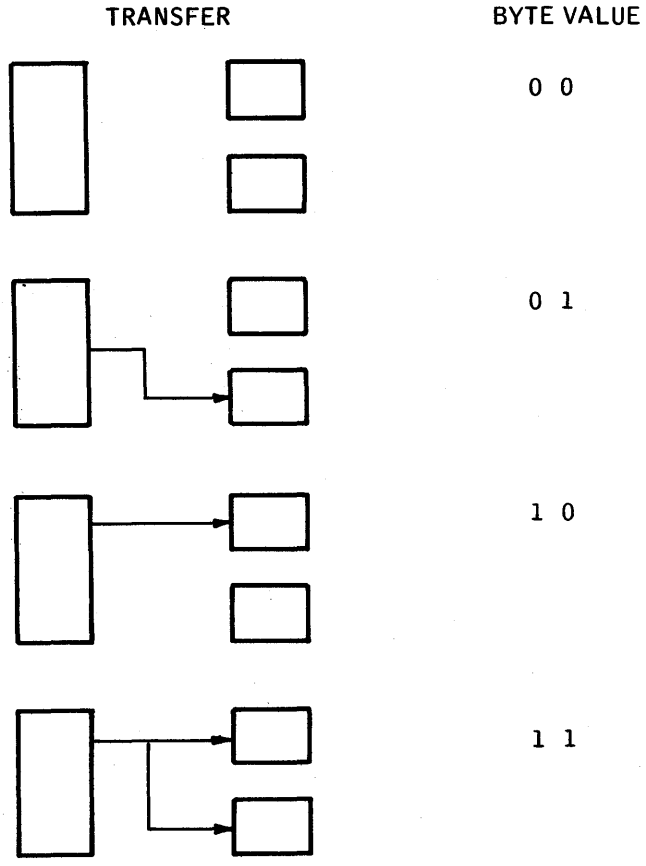


Figure 12—Possible data transfer operations

maps for the splitter is balanced by two advantages of the splitter; i.e., (1) the designer can gets by with only two bits of mapping information in each data word at every stage of the process (this has not been done in Figure 13, but the reader can clearly see why it can be done by looking at Figure 13), and (2) since previously used mapping information is no longer needed, many different indexings can be in process at the same time.

The IPM can be constructed by tracing the desired data output back through the network. Figure 13 is a generalized indexing network for $n = 8$, $m = 4$ constructed using the splitter concept. It is conceivable to combine the networks using only one portion to replace the portions marked AA' and BB', thereby eliminating some transfer hardware and $A(A')$ and $B(B')$ at the expense of more complex clocking and logic. By changing the size of the bytes it is conceivable to construct many different IPM's, but the previously explained IPM seems to be a very good one to use.

This network can be built to provide very high rates of throughput since the level $m/2$ splitter takes half as much time to operate as the m level splitter. With some sophisticated clocking it is conceivable to "time share" the $m/2$ level splitter with two m level splitter and thereby maximize throughput.

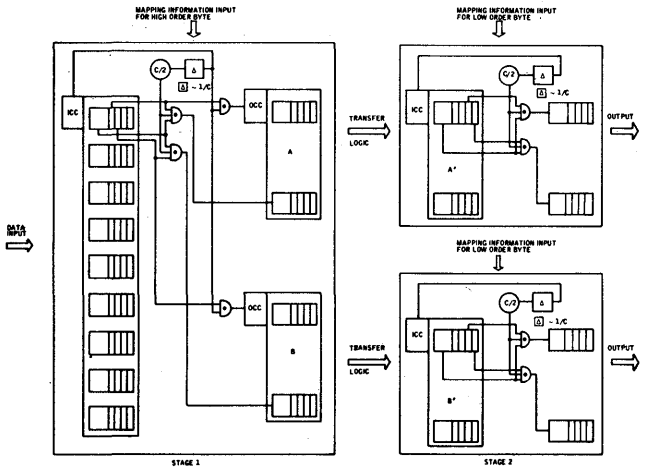


Figure 13—Generalized indexing network

DISTRIBUTED INDEXING NETWORKS

This section presents the final two solutions to the indexing network problem considered in this paper. These two networks are characterized by a highly parallel operation, high speed, and unique timing arrangement. Each network has one comparator (or clocking hardware unit) and one transfer control unit for each word of desired output. In some cases ($m > n + 1$) this requires the addition of several extra blank input registers as in Figure 16. It is assumed that the clock controlling the cycling of the input register has a long enough time between pulses to allow the comparison and transfer of data.

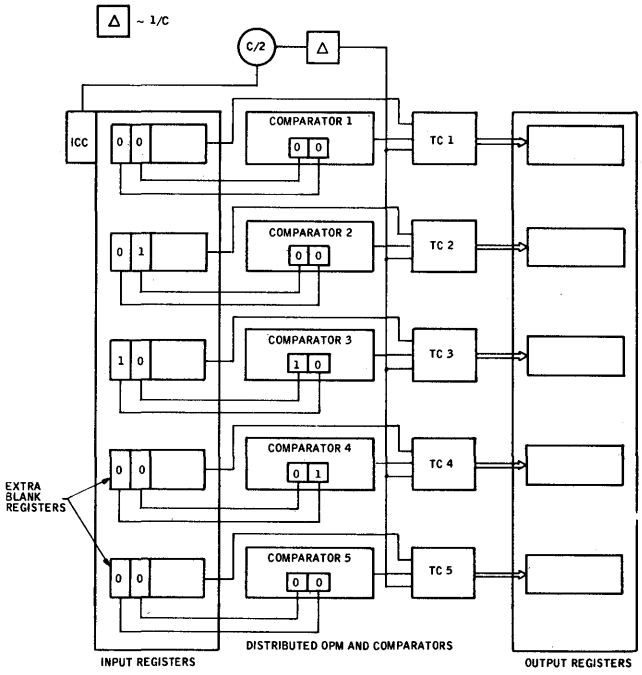


Figure 14—High-speed comparator indexing network set to produce (0 0 X_2 X_1 0)

Both of these networks are based upon the observation that in a complete cycling of the input registers, all data passes through every register. When the correct word is recognized it is immediately transferred.

In the comparator solution in Figure 14, at every clock period the data currently occupying input positions 0, 1, 2, . . . , $M - 1$ is compared to the OPM and the appropriate transfers made. This solution (and the solution shown in Figure 15) requires the larger of m or $n + 1$ clock pulses for the indexing of the input data.

The solution shown in Figure 15 requires a modification of the OPM. The value of the ith position of the OPM is not the binary value of the input data desired, but the number of clock pulses before the input data is in the ith position; i.e., if $Y_i = X_j$ then

$$\text{OPM}(i) = \begin{cases} j - i & \text{if } j \geq i \\ m - (i - j) & \text{if } j < i \text{ and } m > n + j \\ n + 1 - (i - j) & \text{if } j < i \\ & \text{and } n + 1 \geq m \end{cases}$$

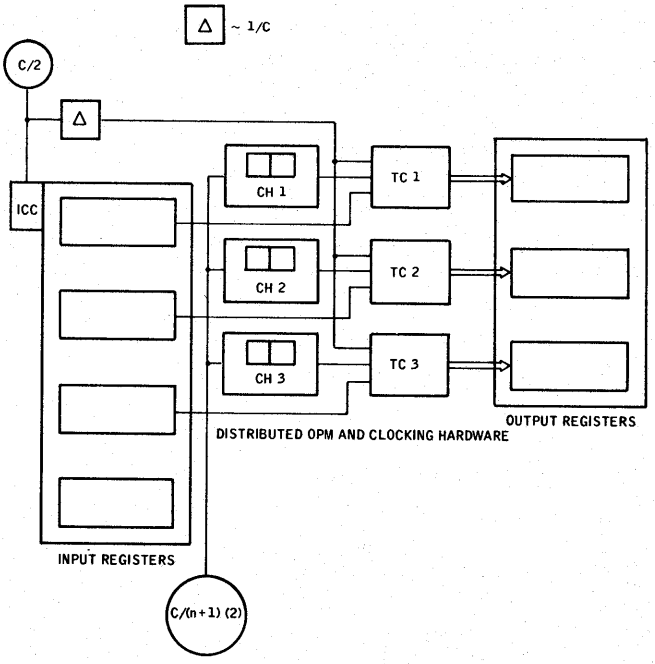


Figure 15—High-speed indexing network

CONCLUSION

A new class of networks was presented in this paper. These networks have the ability to arbitrarily reorder a set of n input cells into m output cells with the repetition or deletion of any cell allowed. Blank cell values may be arbitrarily placed in any of the output cells, thereby allowing the construction of arbitrary contiguous sets of data separated by blanks in the output. These networks have as special cases previously studied permutation and sorting networks. The networks described here are extremely general in nature and should have many different areas of application, particularly, in areas needing networks for routing and transferring data.

The ease of programmability of the indexing networks described is a feature that is extremely unique. Almost all previously studied permutation and sorting networks have long set up and programming times that tend to make them useless in problems in which the destination of the data has to be changed between each set of data inputted. The manner in which the programs are inputted into the network and the simplicity of the program are other features that are unique to the approach followed in this paper. Another unique feature of the solutions presented is the range of tradeoffs they cover. The designer can easily make tradeoff comparisons between the solutions and has many possible different ways to configure each type of network to obtain various speed and hardware comparisons. Hybrid solutions may be extremely attractive. An interesting solution to consider is utilization of the splitter to go from n to $n/2^k$ followed by utilization of 2^k non-splitter networks (like a comparator network). In this manner the large input block of data can be broken down for high-speed "parallel" sorting by other networks.

It is suggested that future research consider construction of high-speed routing networks utilizing the previously described sorting networks. These networks seem to be particularly attractive for the routing and rearranging of data in parallel processors. Another topic that might be of interest is the investigation of the possibilities of performing logic operations on the data while it is being routed (indexed). Consideration might be given to the use of these networks as memories. Some of the logic might be able to be used to convert from a indexing network to a memory.

Some possible applications for the generalized indexing networks are: sorting of data, routing of data, permutation networks, multi-access memories with the number of words of memory accessed a controllable variable, associative memories, multi-access associative memories, reconfigurable multi-processors for real-time users, associative multiprocessors, and any other applications which require the manipulation and reconfiguration of large amounts of data.

BIBLIOGRAPHY

1 K E BATCHER
Sorting networks and their applications
AFIPS Conference Proceedings pp 307–314 SJCC 1968
2 E V BENES
Mathematical theory of connecting networks and telephone traffic
Acadimic Press New York 1965
3 W H KAUTZ K N LEVITT A WAKSMAN
Cellular interconnection arrays
IEEE Trans Electronic Computers Vol C-17 pp 443–451 May 1968
4 A WAKSMAN
A permutation network
Journal of the ACM Vol 15 pp 159–163 January 1968
5 K J THURBER
Design of cellular data switching networks
Submitted for Publication
7 A MUKHOPADHYAY G SCHMITZ K J THURBER K K ROY
Minimization of cellular arrays
Final Report NSF Grant GJ-158 September 1969 Montana State University
Bozeman Montana

The debugging system AIDS

by RALPH GRISHMAN*

New York University

New York, New York

In comparison with the growth of procedural languages over the past decade, the advances in facilities for debugging *compiled code* have been small.[1] The debugging services offered today on most conversational systems have not advanced fundamentally from the design of DDT for the PDP-1.† Batch systems have added a potpourri of other aids—in particular, systems with a machine simulator have included a variety of traces—but, in general, selective tracing and program checks of even slight complexity have been quite messy to invoke, if they were available at all.††

The object of the AIDS project has been to provide a debugging system for FORTRAN and assembly language code on the Control Data 6600 which includes a flexible and reasonably comprehensive set of tools for program tracing and checkout, suitable for both batch and on-line use. A large variety of traces and checks can be invoked through a special "debug language" syntactically similar to FORTRAN. A system of such breadth is really practicable only on a machine with the power and memory capacity of a CDC 6600; such a large debugging system would be difficult to implement on some of the smaller machines on which the earlier interactive debugging systems were developed. At the same time, it is precisely the large, complex programs and supporting systems for machines of this size which make powerful debugging facilities so valuable.

HISTORY

The story of AIDS may be traced back to early 1965, when Prof. J. Schwartz initiated the development of a debugging system for the CDC 6600, which was soon to be delivered to New York University. This system, dubbed the WATCHR, was developed and expanded over the next two years by E. Draughon into a working debugging system.[4] As this system developed, several fundamental difficulties came to light. First, as the options proliferated, calling sequences became more complex, to the point where users not only could not possibly remember the calling sequences, but often would not attempt to invoke some of the more powerful WATCHR features. Second, although WATCHR was adapted for use on the New York University time-sharing system, it was clearly not designed for interactive use. Symbols were not kept at run time, so the user had to refer to his program in terms of absolute addresses; lengthy calling sequences were particularly cumbersome at a teletype.

Thus, in 1967 development was begun on a new debugging system, designed from the outset for conversational as well as batch use, to be invoked through a special procedural language rather than subroutine calls. Design and coding lasted through mid-1968, and distribution of the program began in the spring of 1969.

Several basic requirements were established for the implementation: First, to facilitate maintenance, the same program was to be useable in both batch and interactive modes. Second, to facilitate distribution, the system had to be useable without any modification to the operating system, and have a simple input-output interface adaptable to a variety of environments.

---

* Currently with the Department of Physics of Columbia University, New York, New York.

† The most notable exception of which the author is aware is the debugging system recently developed for TSS/360;[2] readers are referred to this paper for a more detailed discussion of the need for more powerful debugging systems.

†† TESTRAN, the system provided with OS/360 for debugging assembly language routines,[3] includes several features for program checks and conditional traces; however, because the debug commands are macro calls, their format is severely restricted, and consequently test conditions which do not fall into one of several predetermined forms can be quite complicated to encode.

Third, to facilitate use, simple commands had to be provided for the most common debugging requirements.

PROGRAM ORGANIZATION

AIDS, the All-purpose Interactive Debugging System, is a main program with three input files: the object code of the user's program, the listing generated by the compilation (or assembly) of the program, and a "debug file" containing the commands to AIDS for tracing and testing the user's program. AIDS may be divided into three sections corresponding to these three files: the listing reader, which extracts from the compiler and assembler listings the attributes and addresses of the identifiers in the source program; the command translator, which transforms the statements in the debug file into entries in the AIDS trap tables; and the simulator, which simulates, monitors, and traces the user's program.

The listing reader is entirely straightforward, and only one point bears mentioning, namely, that the alternative, modifying the compiler and assembler to output the needed information, was rejected for several reasons. At the time of inception of the project, new FORTRAN compilers were being issued so often by Control Data that reimplementing such a modification on each new compiler would have been a full time effort by itself. In addition, installations with their own compilers would have had to modify them in order to use AIDS, a step many installations might have been hesitant to take.

All debugging information is supplied through a special debug language; absolutely no modifications are required to the user's program to run under AIDS. This debug language will be described in some detail below, after which a few of the techniques used in implementing AIDS will be discussed.

DEBUG LANGUAGE

The three basic syntactic entities of the debug language are the *tag*, the *expression*, and the *event*. The tag designates a fixed location or block of memory, and may be an octal address, statement number, variable name, or subroutine name. The expression specifies a value, and is constructed according to the same rules as a FORTRAN IV expression, including full mixed modes, and logical, relational, and arithmetic operators; only function references are excluded. The event specifies a particular occurrence in the user's program, and can take one of five forms:

OPCODE[S] [⟨opcode⟩ [TO ⟨opcode⟩]]
AT [⟨tag list⟩]
LOAD[S] [[FROM] ⟨tag list⟩]
STORE[S] [[TO] ⟨tag list⟩]
CALLS[S] [⟨tag list⟩]

where ⟨tag list⟩:: = ⟨tag⟩ | (⟨tag⟩ [, ⟨tag⟩] . . .)

In his debug statements, a user can refer to all the identifiers of his source program: variable and subroutine names and statement numbers. Array elements can be referenced with subscripted variables, or the entire array designated by the array name alone (the latter feature is useful, for example, in tracing stores to any element of an array). All hardware registers may be used in arithmetic expressions on an equal footing with other variables. Additional variables may be created at run time for use as counters or switches, and new labels may be assigned to points in the user's program not associated with any identifier in the source text.

The principal statement in the debug language is the trap statement, which has the form

$$\left\{\begin{matrix}\text{WHEN}\\ \text{BEFORE}\\ \text{AFTER}\end{matrix}\right\} \langle\text{event}\rangle, \langle\text{trap sequence}\rangle$$

where ⟨trap sequence⟩:: = ⟨trap command⟩ [, ⟨trap command⟩] . . . This statement directs that immediately before or after (WHEN is synonymous with BEFORE) the occurrence of the specified event* in the simulated program, the commands in the trap sequence are to be executed. The possible trap commands are:

(1) assignment statement—exactly as in FORTRAN
(2) IF (⟨logical expression⟩)

which causes subsequent trap commands to be executed only if the logical expression is true,

(3) SUSPEND {OPCODE | CONTROL | LOAD | STORE | CALL} TRAPS
RESUME {OPCODE | CONTROL | LOAD | STORE | CALL} TRAPS

which turns off/on all traps of a given type,

(4) GO TO ⟨tag⟩

which causes a transfer of control in the simulated program,

(5) PRINT ⟨print item⟩ [, ⟨print item⟩] . . .

where ⟨print item⟩ = ⟨tag⟩ | "⟨text not including"⟩"

which prints the contents of the tag (in a format appropriate to its mode in the user's program), or the specified text, and

* AT ⟨tag⟩ denotes the execution of the first instruction at location ⟨tag⟩. An event without any opcode specification or tag list denotes every opcode, any store, any call, etc.

TAGS

⟨tag⟩:: = ⟨subscripted variable identifier⟩ | ⟨statement identifier⟩ | ⟨global identifier⟩ | ⟨octal identifier⟩

⟨variable identifier⟩:: = ⟨variable name⟩ [$ ⟨subprogram name⟩]

⟨statement identifier⟩:: = ⟨statement number⟩S [$ ⟨subprogram name⟩]

⟨global identifier⟩:: = [$] ⟨global name⟩

⟨subscripted variable identifier⟩:: = ⟨variable identifier⟩ [(⟨subscript⟩[,⟨subscript⟩] . . .)]

EVENTS

⟨event specifier⟩:: = ⟨opcode event specifier⟩ | ⟨control event specifier⟩ | ⟨load event specifier⟩ | ⟨store event specifier⟩ | ⟨call event specifier⟩

⟨opcode event specifier⟩:: = OPCODE[S] [⟨opcode⟩ [TO ⟨opcode⟩]]

⟨control event specifier⟩:: = AT [⟨tag list⟩]

⟨load event specifier⟩:: = LOAD[S] [[FROM] ⟨tag list⟩]

⟨store event specifier⟩:: = STORE[S] [[TO] ⟨tag list⟩]

⟨call event specifier⟩:: = CALL[S] [⟨tag list⟩]

⟨tag list⟩:: = ⟨tag⟩ | (⟨tag⟩ [,⟨tag⟩] . . .)

TRAP STATEMENT

{WHEN / BEFORE / AFTER} ⟨event⟩, ⟨trap sequence⟩

⟨trap sequence⟩:: = ⟨trap command⟩ [,⟨trap command⟩] . . .

TRAP COMMANDS

⟨left side⟩ = ⟨arithmetic expression⟩

where ⟨left side⟩:: = ⟨global identifier⟩ | ⟨variable identifier⟩ [(⟨arithmetic expression⟩ [,⟨arithmetic expression⟩] . . .)]

IF (⟨logical expression⟩)

SUSPEND ⟨trap type⟩ [⟨trap word⟩]

RESUME ⟨trap type⟩ [⟨trap word⟩]

where ⟨trap type⟩:: = OPCODE | CONTROL | LOAD | STORE | CALL

⟨trap word⟩:: = TRAP[S] | TRACE[S]

PRINT ⟨print element⟩ [,⟨print element⟩] . . .

where ⟨print element⟩:: = ⟨tag⟩ | "⟨text not containing"⟩"

COMMANDS NOT VALID IN TRAP SEQUENCES

WHAT [IS] ⟨tag⟩

WHERE [[IS] ⟨tag⟩]

$⟨subprogram name⟩ *

RETREAT ⟨decimal integer⟩

LABEL ⟨octal identifier⟩ **

MAP {ON / OFF / TRACE}

STEP {⟨decimal integer⟩ / OFF}

= ⟨variable identifier⟩ = ⟨arithmetic expression⟩ ***

BREAK OUT[AT ⟨tag⟩]

BREAK IN AT ⟨tag⟩

---

* establishes new local (default) subprogram for identifiers

** finds nearest symbolic label

*** defines (assigns an address to) a symbol

Figure 1—The syntax of the AIDS debug language

(6) TRACE

which prints a line describing the event which caused the trap. Since the statement

WHEN ⟨event⟩, TRACE

is one of the most often used, the natural abbreviation

TRACE ⟨event⟩

has been allowed. A few sample trap statements:

```
TRACE STORES TO A
WHEN CALL TEST, IF (I**3+J . GT . 27)
  PRINT "INVALID ARGUMENTS TO TEST,"
                                  I, J
WHEN AT 10S, I=I+1, IF(I . GT . 100) PRINT
"LOOP EXECUTED 100 TIMES, EXIT FORCED";
                                  GO TO 100S
```

(the S after the numbers in the last statement indicate that they are statement numbers rather than integers).

The user has at his disposal quite a few other control and informational commands; these are enumerated in Figure 1. A few of these deserve special mention:

The MAP feature provides a simple means of tracing the flow of control in his program. The MAP TRACE command makes AIDS print out pairs of addresses between which instructions were executed without any transfers. If the user does not want a continuous map, he can still get the last 25 such pairs printed at any time by typing MAP.

The user can step forward through his program, a fixed number of instructions at a time, with the command

STEP ⟨integer⟩

More interestingly, with the command

RETREAT ⟨integer⟩

he can step *backwards* through his program by a fixed number of instructions; his program is restored to exactly the same status it had earlier. Although this process is limited to a few thousand instructions, it is

The program to be debugged:

```
      PROGRAM PRIME (INPUT, OUTPUT)
C     PROGRAM DETERMINES IF A NUMBER IS
        PRIME
   10 PRINT 20
   20 FORMAT(*YOUR NUMBER, PLEASE-*)
      READ 40, NUM
   40 FORMAT(I4)
      IF (NUM. LE. 0) CALL EXIT
      ISQRT = NUM ** (½) + .1
   70 DO 90 J = 2, ISQRT
   80 IF (NUM/J*J. EQ. NUM) GO TO 130
   90 CONTINUE
      PRINT 110
  110 FORMAT (* NUMBER IS PRIME*)
      GO TO 10
  130 PRINT 140
  140 FORMAT (* NUMBER IS NOT PRIME*)
      GO TO 10
      END
```

A log of the debug session:	Explanation:
TYPE PROGRAM NAME–lgo	
BEHEST–when at 10s, pause	pause each time before first print
BEHEST–go	start execution
PAUSE.	have reached statement 10
BEHEST–go	keep going
YOUR NUMBER, PLEASE–9	try program with 9
NUMBER IS PRIME	program doesn't work
PAUSE.	are back at statement 10
BEHEST–trace stores to j	watch DO-loop index
BEHEST–go	and try again
YOUR NUMBER, PLEASE–9	
STORE TO J = 2	why didn't it try J = 3?
NUMBER IS PRIME	
PAUSE.	
BEHEST–what is isqrt?	check limit of DO loop
1	aha! realize that formula for ISQRT is wrong
BEHEST–after store to isqrt, fnum = num, isqrt = fnum** 0.5 + .1	fix it
BEHEST–go	
YOUR NUMBER, PLEASE–9	
STORE TO J = 2	
STORE TO J = 3	
NUMBER IS NOT PRIME	seems to work now
PAUSE.	
BEHEST–go	try one more
YOUR NUMBER, PLEASE–5	
STORE TO J = 2	
NUMBER IS PRIME	
PAUSE.	
BEHEST-quit	

NOTE: Input from user appears above in lower case; output from AIDS appears in standard upper case; output from the user's program appears in italicized upper case. The comments on the right would not be a part of an actual debugging session. "BEHEST–" is the prompt given by AIDS when it expects input

Figure 2—A trivial example of an on-line session with AIDS

generally very helpful in determining the source of difficulty when an error condition occurs.

Programs running under AIDS are normally simulated rather than executed directly; the trap commands described above are in effect only while the program is being simulated. Simulation, however, greatly increases the time required for program execution (by a factor of 60 or more), so that programs which run into difficulties only after several minutes of execution cannot be debugged by simulation alone. For users who believe they can localize the source of their difficulties, or who only require the AIDS trap facilities at specific points in their program, the commands

BREAK OUT AT ⟨tag⟩

and

BREAK IN AT ⟨tag⟩

have been provided. These commands direct AIDS to change from simulation to direct program execution, and to revert to simulation at arbitrary points in a program.

To illustrate the use of a few of these commands in the interactive mode, a trivial debugging example is given in Figure 2.

INTERNAL DESIGN

The most important decision in designing a debugging system is whether to process the source language directly (by adding debugging statements to a compiler, or interpreting the source text) or to work from the object code. The author firmly believes that both types of debugging aid should be included in standard programming support, particularly for time-sharing systems. A system using only the object code and symbol table of a program cannot offer the simplicity of code modification possible with an interpreter or incremental compiler; nor can it provide several types of error checking which are easy to perform at

the source language level, such as subscript in range and agreement in type of formal and actual parameters. However, several considerations dictated development of a system running from the object code. First, a large fraction of users have assembly language subroutines in their FORTRAN programs; running such programs interpretively would in effect mean assembling the source code and then simulating the machine instructions. Second, some of the most elusive bugs are due to compiler and system routine errors; such bugs can clearly only be found by a system which runs from the compiled code. (Interestingly enough, some of the first bugs found by AIDS were in the compiler, loader, FORTRAN coded output routine, and the routine which generates FORTRAN execution-time error messages.)

The next choice to be made is whether to simulate or execute the object code. In contrast to most debugging systems, AIDS offers the user the ability to do either. Simulation provides a far richer set of traces and checks than could a system which executes the object code; in particular, it provides a simple solution to what appears to be the most common plight of the desperate user, "What part of my program stored *that*?" On the other hand, when a particular routine can be isolated as the source of a program error, only that routine need be simulated, with the rest of the code executed; in this case, the program can run at nearly normal speed.

The trap system is entirely straightforward, using for each type of trap (load, store, etc.) a list of addresses which is checked regularly during simulation. To avoid possible "side-effects" (e.g., instruction modification at a location where a breakpoint is stored) absolutely no modifications are made to the user's program during simulation. Trap commands are checked syntactically and translated into an internal form on input, and are interpreted whenever a trap occurs.

Whenever a store is performed by the simulated program, the old contents of the referenced memory location are saved in a circular buffer. At two points in the circuit of the circular buffer, the contents of all the simulated hardware registers are saved. When a RETREAT is requested, the user's program is first reset to its status at one of these two earlier points; memory is restored by working backwards through the circular buffer from its current position to the earlier point. The program is then stepped forward to the point to which the user wanted to retreat in the first place.

AIDS consists of about 6000 source cards, and occupies a minimum of 44000_8 words of memory. With the exception of the simulation routine, which was coded in assembly language for efficiency, the entire system was written in FORTRAN. This was no doubt a factor in getting the system coded and largely debugged in less than one man-year of programming effort.

CONCLUSION

In evaluating the results of the AIDS project, it is necessary to ask two separate questions: Is such a powerful debugging system worthwhile? and Has this implementation been successful, in particular with respect to the three points mentioned towards the beginning of this paper?

The latter question I believe can be answered in the affirmative; as regards the three specific points:

1. The identical program has been used for both batch and conversational debugging. In general the system appears to be flexible enough to satisfy the debugging styles of both types of user: the selective traces and automatic program checks required by the batch user and the conditional trapping desired by the time-sharing user.*

2. In large part because most of AIDS is coded in FORTRAN, it has been converted for use under two batch and three conversational systems with relative ease. In addition, the modular design has made it possible for the author of a subsequent CDC 6600 debugging system to incorporate major sections from AIDS.[5]

3. The only "abbreviation" included is the TRACE command (in place of WHEN . . . , TRACE). Short of a general redesign of the command structure to reduce the amount of typing required, no other particular sequences of commands seemed to be frequent enough to merit abbreviation.

The more general question, whether such a powerful debugging system is worth the cost, is more difficult to answer. There is, of course, the increased cost in processor time and memory space, but these items generally represent only a small part of the cost of debugging; as these costs decrease further, it is safe to assume that nearly any significant saving of a programmer's time at the expense of computer time will represent a net savings.

Thus the fundamental question is, does AIDS save the programmer time in debugging? In one aspect it clearly does not: since it is such a large system, it takes quite a while to learn all its capabilities. Indeed,

* It has been suggested that the ability to jump around within the deck of commands to AIDS may be desirable to give the batch user even greater control over the debugging process; such a facility may soon be added.

potential one-time or occasional users have been dissuaded by the thought of reading a 23-page manual. As a result, most AIDS users until now have been systems programmers or user consultants. One user has suggested, however, that it is precisely these experienced users who are most in need of such a system and for whom the system should be designed; in this case the time required to become familiar with the system is not such a critical factor.

So, finally: Does AIDS save time in the actual task of debugging, in comparison with simpler debugging systems? In the batch mode, where the primary object is to collect as much useful information as possible from each run, I am confident that the answer is yes. In conversational debugging, on the other hand, brevity and ease of typing are important factors; these aspects clearly favor the simple debugging systems, where considerable effort has been expended in this area,[6] over the syntactically complex AIDS. It is the author's impression, however, that the few most difficult program bugs—those in which a very powerful system like AIDS can be expected to be the most help—are the ones which consume most of a programmer's time and cause most of his ulcers. In any event, a good deal more experience with the on-line use of AIDS and similar debugging systems will be required to find the best balance of brevity, simplicity, and power.

REFERENCES

1 T G EVANS D L DARLEY
On-line debugging techniques: A survey
FJCC Proceedings 1966

2 W A BERNSTEIN J T OWENS
Debugging in a time-sharing environment
FJCC Proceedings 1968

3 *System/360 operating system TESTRAN*
IBM Form No C28-6648-1

4 E DRAUGHON
WATCHR III—A program analyzing and debugging system for the CDC 6600, user's manual
AEC Research and Development Report NYO-1480-58

5 H E KULSRUD
Helper—An interactive extensible debugging system
IDA—Communications Research Division Working Paper No 258

6 P T BRADY
Writing an on-line debugging program for the experienced user
CACM Vol 11 No 6 p 423 June 1968

Sequential feature extraction for waveform recognition

by W. J. STEINGRANDT* and S. S. YAU

Northwestern University
Evanston, Illinois

INTRODUCTION

Many practical waveform recognition problems involve a sequential structure in time. One obvious example is speech. The information in speech can be assumed to be transmitted sequentially through a phonetic structure. Other examples are seismograms, radar signals, or television signals. We will take advantage of this sequential structure to develop a means of feature extraction and recognition for waveforms. The results will be applied to speech recognition.

An unsupervised learning (or clustering) algorithm will be applied as a form of data reduction for waveform recognition. This technique will be called *sequential feature extraction.* The use of sequential feature extraction allows us to represent a given waveform as a sequence of symbols $a_{\sigma 1}, \cdots, a_{\sigma k}$ from a finite set $A = \{a_1, \cdots, a_M\}$. This method of data reduction has the advantage of preserving the sequential structure of the waveform. The problem of waveform recognition can be transformed into a vector recognition problem by expanding the waveform using orthogonal functions.[1] However, in this case the sequential structure is masked because the expansion operates on the waveform as a whole. Data reduction can also be carried out by time sampling and storing the samples as a vector. In this case the dimension of the vector is usually large. The data produced by sequential feature extraction is more compact. We will formalize the concept of sequential feature extraction and develop a performance criterion for the resulting structure. An unsupervised learning algorithm, which will optimize this structure with respect to the performance criterion, is presented. This algorithm, which can be applied to waveform recognition as well as vector recognition, represents an improvement over existing clustering algorithms in many respects. This method will allow unbounded strings of sample patterns for learning. The samples are presented to the algorithm one at a time so that the storage of large numbers of patterns is unnecessary.

The assumption of known probability measures is extremely difficult to justify in most practical cases. This assumption has been made in a number of papers,[2–5] but no such assumption is made here. That is, the requirement for convergence is only that the measures be smooth in some sense. Braverman's algorithm[6] has been shown to have these advantages. However, he assumes that there are only two clusters, which, after a suitable transformation, can be strictly separated by a hyperplane. These assumptions are too restrictive for the practical applications considered in this work. In the clustering algorithm to be presented here, any number of clusters is allowed, the form of the separating surfaces is not as restricted, and strict separability of the clusters is not assumed. This algorithm is considerably more general than existing clustering algorithms in that it applies to time varying as well as time invariant patterns.

We will assume that the waveform is vector valued, i.e., $\mathbf{x}(t)$ is in a set $\Omega = \{\mathbf{x}(t) \mid ||\dot{\mathbf{x}}(t)|| < M$, all $t \in [0, T_x]\}$, where $\dot{\mathbf{x}}(t)$ is the componentwise time derivative of $\mathbf{x}(t)$. It is assumed that each pattern class has some unknown probability measure on this set.

A unified model for waveform recognition and vector recognition will be presented. It will be shown that the recognition of a vector pattern can be considered as a special case of waveform recognition. This will be done by observing that the pattern space of n-vectors $\mathbf{v}$ is isomorphic to the space of all constant functions $\mathbf{x}(t) = \mathbf{v}$.

Recognition of real functions of time will be possible by defining a transformation to the space Ω or by assuming that $\mathbf{x}(t)$ is one-dimensional. The problem of waveform recognition will be carried out in the space Ω, where the dimension of $\mathbf{x}(t)$ is most likely greater than one.

The experiments on speech will show an interesting relationship between the sequential features and the

* Presently with IBM Corporation, Rochester, Minnesota.

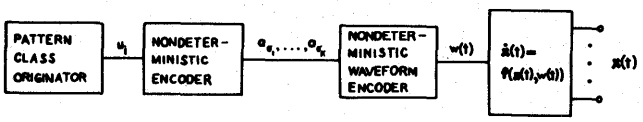


Figure 1—Assumed process producing pattern waveforms

standard linguistic phonetic structure for English. A recognition algorithm using sequential machines will be given that will accept symbol strings $a_{\sigma 1}, \cdots, a_{\sigma k}$ to classify spoken words.

SEQUENTIAL FEATURE EXTRACTION

Figure 1 shows the process that is assumed to produce the vector waveform $\mathbf{x}(t)$. It is emphasized that this model may not represent an actual physical process as described. It is included as a means of demonstrating the assumptions about the sequential structure on Ω. In the figure it is assumed that there is some state of nature or intelligence such that pattern class i is present. The pattern classes are represented by the symbols u_i, $i = 1, \cdots, R$. There exists a second set of symbols $A = \{a_1, \cdots, a_M\}$ called the *phoneme set.* Each a_i is called a *phoneme* (while the terminology is suggestive of speech and language, there may be little relation to the speech recognition problem). The second step converts u_i into a finite sequence of phonemes $a_{\sigma 1}, \cdots, a_{\sigma k}$, where σ_i is the index of the ith phoneme in the sequence. The process of encoding u_i into $a_{\sigma 1}, \cdots, a_{\sigma k}$ is most likely unknown and is probably nondeterministic. That is, the sequence generated by a given u_i may not be unique.

Each sequence is then assumed to go through an encoding process into a real waveform $w(t) \in W$, where W is the set of all continuously differentiable real waveforms such that $\dot{w}(t)$ and the time duration are bounded. This process is also most likely nondeterministic. For the most part, this encoding process is unknown; but some assumptions can be made. It is assumed that there is some unique behavior of $w(t)$ for each a_i. As each $a_{\sigma i}$ from $a_{\sigma 1}, \cdots, a_{\sigma k}$ is applied to the encoder, the behavior of $w(t)$ changes in some manner. This behavior is detected by using a transformation to a vector function of time $\mathbf{x}(t) \in \Omega$. This transformation can be considered to be described by some differential equation of the form

$$\dot{\mathbf{x}}(t) = \mathbf{f}[\mathbf{x}(t), w(t)], \tag{1}$$

where $f: R^n \times R \to R^n$ is a bounded continuous function. The explicit form for this equation may not be known, but the system that it describes is assumed to be determined from the physical process producing $w(t)$. If this differential equation is properly chosen, then the value of $\mathbf{x}(t)$ at any time t is some pertinent measure of the recent past behavior of $w(t)$.

We will shortly present a clustering algorithm on Ω which is a generalization of the usual concept of clustering. Clustering for the time invariant case will first be reviewed. It is assumed that there exists a metric ρ that measures the similarity between patterns, where the patterns are assumed to be fixed points in R^n. ρ is such that the average intra-class distance is small, while the average inter-class distance is large. The method of cluster centers used by Ball and Hall[7] will be used to detect the clusters. It is assumed that the number of clusters is fixed, say at M, and there are $s_i \in R^n$, $i = 1, \cdots, M$, such that each $\mathbf{s}_i$ has minimum mean distance to the points in its respective cluster. These $\mathbf{s}_i$ can be found by minimizing the performance criterion $E_x \min_i \rho(\mathbf{s}_i, \mathbf{x})$, where the expectation is with respect to the probability measure on R^n.

These assumptions will now be generalized for patterns that are time varying. Here the phonemes a_i play the part of the pattern class for the time invariant case. That is, the time invariant pattern vectors are assumed to be the same as the time varying case except that the phoneme sequence producing the vector is always of length one, and $\mathbf{x}(t)$ is the constant function.

We will describe the general case in more detail. Here, as before, it is assumed that there is a similarity metric ρ on R^n. This metric measures the similarity of the behavior of $w(t)$ at any given time t_1 to that at any other time t_2. This is done by measuring the distance $\rho[\mathbf{x}(t_1), \mathbf{x}(t_2)]$, where it is understood that $\mathbf{x}(t)$ and $w(t)$ satisfy (1). The assumption is that (1) and ρ are such that if a_i was applied to the waveform encoder both at time t_1 and t_2, then $\rho[\mathbf{x}(t_1), \mathbf{x}(t_2)]$ is small. On the other hand, if a_j was applied during t_1 and a_i during t_2, then $\rho[\mathbf{x}(t_1), \mathbf{x}(t_2)]$ is large for $i \neq j$. In other words, each a_i produces behavior in $w(t)$ such that the corresponding values for $\mathbf{x}(t)$ tend to cluster in distinct regions of R^n. Thus, the a_i are represented by clusters in R^n. It is assumed that each a_i has a cluster center $\mathbf{s}_i$ associated with it. This implies that for each a_i there is a point $\mathbf{s}_i \in R^n$ such that when a_i is applied to the waveform encoder, the function $\mathbf{x}(t)$ tends to pass close to $\mathbf{s}_i$.

It will also be assumed that $\mathbf{x}(t)$ spends most of its time in those regions that are close to the $\mathbf{s}_i$. In other words, the more important features of $w(t)$ are of longer duration. The example shown in Figure 2 illustrates the foregoing assumptions. The figure shows the action of $\mathbf{x}(t)$ under the application of a_1, a_2, a_3 to the

encoder. In the figure the width of the path is inversely proportional to $||\dot{\mathbf{x}}(t)||$.

This model is necessarily somewhat vague because we are unwilling to make assumptions about the probability measures on Ω. If such assumptions were made, then a more formal definition of a cluster might be possible. For most practical problems such as speech recognition, these types of assumptions cannot be made.

Assuming ρ and the $\mathbf{s}_i$ were known, they could be used to reconstruct an estimate of the sequence $a_{\sigma 1}, \cdots, a_{\sigma k}$ for an unknown waveform $\mathbf{x}(t)$ in the following manner. Referring to Figure 3, each of the quantities $\rho[\mathbf{s}_i, \mathbf{x}(t)]$, $i = 1, \cdots, M$ are continuously calculated and the minimum continuously indicated. That is, suppose there exist times $t_1 = 0, t_2, \cdots, t_{k+1} = T_x$ such that $\rho[\mathbf{s}_{\sigma i}, \mathbf{x}(t)] \leq \rho[\mathbf{s}_j, \mathbf{x}(t)]$ for all $j \neq i$ and all $t \in [t_i, t_{i+1}]$, then it is assumed that the phoneme sequence most likely to have produced $\mathbf{x}(t)$ is $a_{\sigma 1}, \cdots, a_{\sigma k}$. Note that no adjacent phonemes in the sequence are ever the same. It is also apparent that the output sequence is independent of time scale changes in $\mathbf{x}(t)$. If ρ and (1) are fixed, then for a given set of the $\mathbf{s}_i$, $i = 1, \cdots, M$ there is a transformation defined by Figure 3. This transformation will be called $T_s: \Omega \rightarrow P$, where P is the set of all finite sequences of symbols from A, $\mathbf{s} = (\mathbf{s}_1', \cdots, \mathbf{s}_M')'$ and the prime of a matrix denotes its transpose.

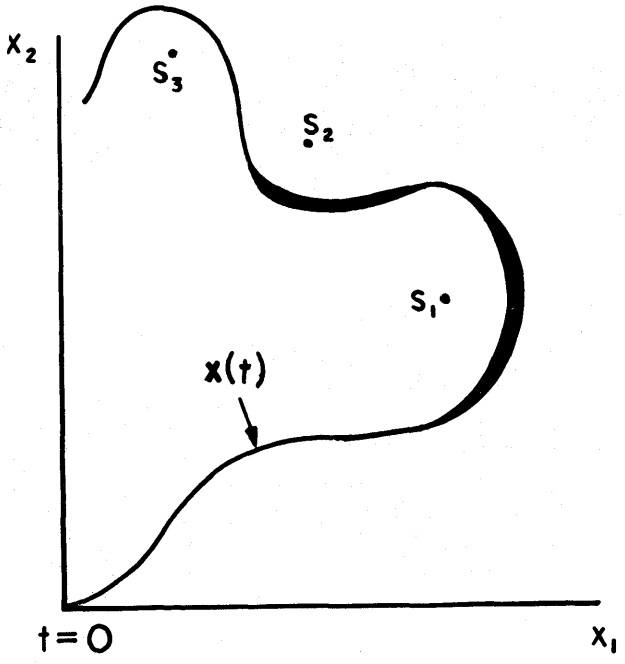


Figure 2—Example of waveform $x(t)$ produced by sequence a_1, a_2, a_3

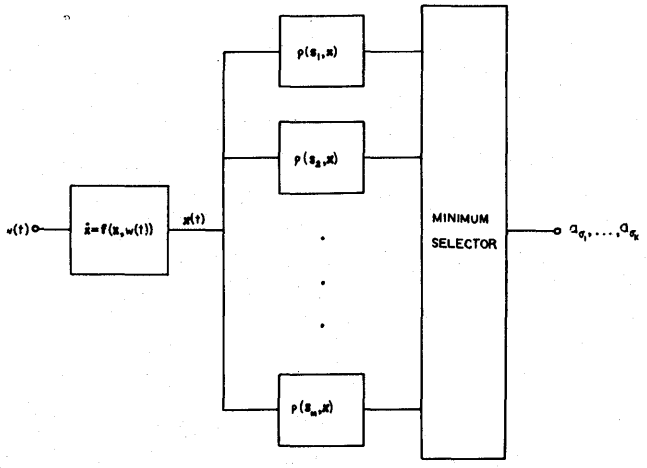


Figure 3—Implementation of a phonetic structure

The pair (A, T_s) defines a sequential structure on Ω. This sequential structure is extracted by the transformation T_s defined in Fig. 3. Thus, the terminology *sequential feature extraction* has been used.

This definition of sequential feature extraction is unique in that it puts sequential structures in waveform recognition on a more formal basis. Gazdag[8] has suggested a somewhat similar structure in what he calls *machine events*. His method involves linear discriminant functions, and he gives no method for determination of the structure.

The objective of the learning algorithm will be to determine (A, T_s) by determining the composite vector $\mathbf{s}$. The differential equation in (1) and ρ are assumed to be determined from a study of the physical process producing $w(t)$. It is obvious that any random choice for $\mathbf{s}$ will define a sequential structure. The learning algorithm will be required to find that $\mathbf{s}$ which is optimum with respect to some performance function. This performance function is generalized from that mentioned previously for time invariant patterns.

Based on the previous discussion, the performance function for this case is

$$E_x C(\mathbf{s}, \mathbf{x}) = E_x\left(\frac{1}{T_x}\int_0^{T_x} \{\min_i \rho[\mathbf{s}_i, \mathbf{x}(t)]\}\, dt\right), \quad (2)$$

where $C(\mathbf{s}, \mathbf{x})$ is a function called the *confidence function* for a given waveform $\mathbf{x}(t)$. The smaller $C(\mathbf{s}, \mathbf{x})$ is for a given $\mathbf{x}(t)$, the more confidence, on the average, can be placed in the resulting sequence of phonemes. Taking the statistical expectation over the entire population Ω gives us the performance function.

The object of the learning rule will be to find an $\mathbf{s}^*$ such that $E_x C(\mathbf{s}^*, \mathbf{x})$ is at least a local minimum for

$E_x C(\mathbf{s}, \mathbf{x})$. It is obvious from (2) that direct evaluation of the performance function is not possible because the probability measures are not known. Using stochastic approximation, it can be shown that if a learning rule of the form

$$\mathbf{s}^{n+1} = \mathbf{s}^n - a_n \nabla C(\mathbf{s}^n, \mathbf{x}^n) \tag{3}$$

is used, then under certain conditions the sequence $\{\mathbf{s}^n\}$ converges almost surely to a saddle point or local optimum $\mathbf{s}^*$, where $\mathbf{s}^n$ is the value for $\mathbf{s}$ at the nth stage of learning, $\mathbf{x}^n$ is the nth sample waveform, and a_n is a sequence of scalers satisfying certain convergence conditions. Note that $\mathbf{x}^n$ is unlabeled, i.e., no pattern class information is used in the learning rule.

It can easily be seen that if $\mathbf{x}(t) = \mathbf{v}$, and $T_x = 1$, then the performance function in (2) reduces to that for the time invariant case.

We are now in a position to calculate $\nabla C(\mathbf{s}, \mathbf{x})$ for a given pattern $\mathbf{x}(t)$. Define

$$A(\mathbf{s}_i) = \{\mathbf{x} \in R^n \mid \rho(\mathbf{s}_i, \mathbf{x}) < \rho(\mathbf{s}_j, \mathbf{x}), \quad \text{all } j \neq i\}. \tag{4}$$

Each region $A(\mathbf{s}_i)$ corresponds to a phoneme a_i. For each $\mathbf{x}(t)$, the sequence $a_{\sigma 1}, \cdots, a_{\sigma k}$ is simply a list of the regions $A(\mathbf{s}_i)$ through which $\mathbf{x}(t)$ passes. The $t_1, \cdots, t_{k+1}$ are then the times at which $\mathbf{x}(t)$ passes from one region to the next. Using this, we can write

$$C(\mathbf{s}, \mathbf{x}) = \sum_{i=1}^{k} \int_{t_i}^{t_{i+1}} \rho[\mathbf{s}_{\sigma i}, \mathbf{x}(t)]\, dt \tag{5}$$

Taking the gradient and canceling terms we have

$$\nabla_{\mathbf{s}_j} C(\mathbf{s}, \mathbf{x}) = \sum_{\sigma i = j} \int_{t_i}^{t_{i+1}} \nabla_{\mathbf{s}_j} \rho[\mathbf{s}_{\sigma i}, \mathbf{x}(t)]\, dt \tag{6}$$

where $\nabla_{\mathbf{s}_j}$ is the gradient with respect to $\mathbf{s}_j$. It is also understood that the integral of a vector function is meant to be the vector of integrals of each of the individual components. The learning rule in (3) becomes

$$\mathbf{s}_j^{n+1} = \mathbf{s}_j^n - \frac{a_n}{T_{x_n}} \sum_{\sigma i = j} \int_{t_i}^{t_{i+1}} \nabla \mathbf{s}_j \rho[\mathbf{s}_{\sigma i}{}^n, \mathbf{x}_n(t)]\, dt \tag{7}$$

where

$$\sum_{n=1}^{\infty} a_n = \infty, \qquad \sum_{n=1}^{\infty} a_n{}^2 < \infty, \tag{8}$$

$\mathbf{x}_n(t)$ is the nth sample waveform, and $\mathbf{s}_j{}^n$ is the value of $\mathbf{s}_j$ at the nth step of learning. An equivalent form is

$$\mathbf{s}_j^{n+1} = \mathbf{s}_j^n - a_n \int_0^1 \chi_j[\mathbf{x}_n(t)] \nabla_{s_j} \rho[\mathbf{s}_j, \mathbf{x}_n(t)]\, dt, \tag{9}$$

where χ_j is the characteristic function of $A\ \mathbf{s}_j$).

Example 1 Assume that ρ is the squared euclidean metric, i.e.,

$$\rho(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} (x_i - y_i)^2.$$

The learning rule in this case becomes

$$\mathbf{s}_j{}^{n+1} = \mathbf{s}_j{}^n - \frac{a_n}{T_{x_n}} \int_0^{Tx_n} \chi_i[\mathbf{x}_n(t)][\mathbf{s}_i{}^n - \mathbf{x}_n(t)]\, dt. \tag{10}$$

AUTOMATIC SPEECH RECOGNITION

The automatic recognition of speech has received much attention since the advent of the digital computer. Most of the previous work[9–12] in speech recognition has made use of the phonetic structure of speech. Almost all of these studies use the standard linguistic phonetic structure. Here we investigate the applicability of sequential feature extraction to the speech recognition problem. A sequential structure will be developed using a limited vocabulary. It will be seen that the resulting structure is related to the standard English phonetic structure. Because of this relationship to speech, we will refer to sequential feature extraction as a *machine phonetic structure.*

In order to represent the speech waveform $w(t)$ as a vector function of time we will use the common method[13] of a bank of bandpass filters. In the experiments 50 filters were spaced from 250 to 7000 hz. Each filter was envelope detected and sampled by an A/D converter and multiplexor. Therefore, $\mathbf{x}(t)$ is a 50 dimensional vector function of time.

Kabrisky[14] has shown that a neuron network similar to that found in the brain is capable of performing correlation calculations. Based on this we assume that the similarity metric defined by

$$\rho(\mathbf{x}, \mathbf{y}) = \left(1 - \frac{\mathbf{x} \cdot \mathbf{y}}{\| x \| \| y \|}\right) = \frac{1}{2}\left(\frac{\mathbf{x}}{\| \mathbf{x} \|} - \frac{\mathbf{y}}{\| \mathbf{y} \|}\right)^2 \tag{11}$$

is valid for speech sounds. Note that $\rho(a\mathbf{x}, b\mathbf{y}) = \rho(\mathbf{x}, \mathbf{y})$ for all a, b, $\mathbf{x}$, $\mathbf{y}$, i.e., the metric ρ is invariant to amplitude changes in the signal. Using this metric we have the following learning rule.

$$\mathbf{s}_i{}^{n+1} = \mathbf{s}_i{}^n - a_n \mathbf{\Delta}_i, \tag{12}$$

where

$$\mathbf{\Delta}_i = \frac{1}{\| \mathbf{s}_i{}^n \|}\left(\frac{1}{\| \mathbf{s}_i{}^n \|^2} \mathbf{s}_i{}^n \mathbf{s}_i{}^{n\prime} - \mathbf{I}\right) \frac{1}{T_{x_n}} \int_0^{Tx_n} \chi_i[\tilde{\mathbf{x}}_n(t)] \tilde{\mathbf{x}}_n(t)\, dt \tag{13}$$

where

$$\tilde{\mathbf{x}}_n(t) = \frac{1}{||\mathbf{x}_n(t)||^2}\mathbf{x}_n(t), \tag{14}$$

and $\mathbf{I}$ is the $n \times n$ identity matrix.

If we normalize $\mathbf{x}(t)$ as part of the preprocessing and normalize each $\mathbf{s}_i$ after each step of learning, then we can write the learning rule as

$$s_i^{n+1} = \mathbf{s}_i^n - a_n[\mathbf{s}_i^n\mathbf{s}_i^{n'} - \mathbf{I}]T_{x_n}^{-1} \int_0^{Tx_n} \chi_i[\mathbf{x}(t)]\mathbf{x}(t)\,dt \tag{15}$$

This rule was used to develop the phonetic structure presented in the next section on the experimental results.

MACHINE PHONETIC STRUCTURE EXPERIMENTAL RESULTS

This section describes the results of experiments using the data acquisition equipment previously described. The basic goals of the experiments were

(1) test convergence of the algorithm
(2) determine effects of local optimums
(3) provide output for use in speech recognition
(4) determine relationship to the standard linguistic phonetic structure, if any.

There were two sets of data used for the tests. One set consisted of 50 utterances each of the words "one", "four", "oaf", "fern", "were". These words were chosen because they contained a small number of sounds with an unvoiced as well as voiced sounds. One speaker was used for all utterances. It was found that the speaker's voice had enough variation to adequately test the algorithm. If the algorithm had been tested with many speakers, the variance would have been much larger. This would have lengthened the convergence times beyond what was necessary for a sufficient test.

The larger data set consisted of 40 utterances of each of the ten digits "one", "two", . . ., "nine", "oh". These were all spoken by the same person. These words contain a wide variety of sounds: voiced, unvoiced, dipthongs, plosives, etc. This set was used to give a somewhat more severe test of convergence and to provide data for speech recognition. We will now consider the four goals of the experiments separately.

Convergence: Many runs with the small data set were made. Different starting points were chosen, and other conditions were varied. In all cases the algorithm showed a strong convergence.

Because there was only a finite set of samples, the convergence properties in (8) were academic. In order to better determine convergence, the sequence $\{a_n\}$ was chosen to be constant over many steps of learning. If convergence was apparent under these conditions, then convergence under decreasing step increments can be assumed.

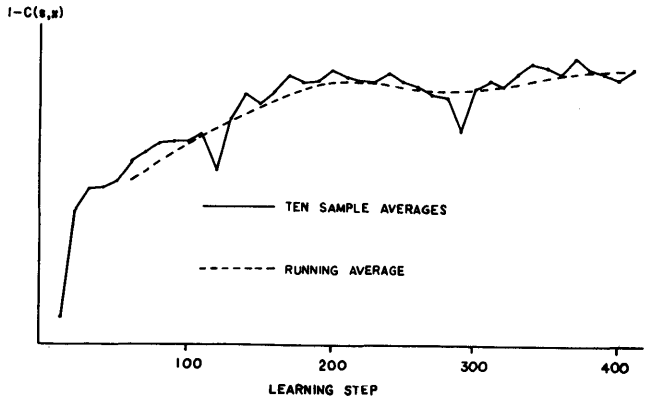


Figure 4—Improvement of the performance function

Figure 4 shows an example of the convergence of $C(\mathbf{s}, \mathbf{x})$ using the large data set. Due to the variance of the data, a direct plot of $C(\mathbf{s}, \mathbf{x})$ at each step of learning shows very little. The individual points for $C(\mathbf{s}, \mathbf{x})$ are so scattered that convergence is difficult to see. Figure 4 shows the plot after data smoothing. The solid curve represents averages of ten successive values of $C(\mathbf{s}, \mathbf{x})$. The dotted line represents further data smoothing. It can be seen that the performance function is not improved at each step but is improving over many samples. In order to demonstrate that the components of $\mathbf{s}^n$ were converging as well as $C(\mathbf{s}, \mathbf{x})$, the plot in Figure 5 was made. This is a plot of the tenth

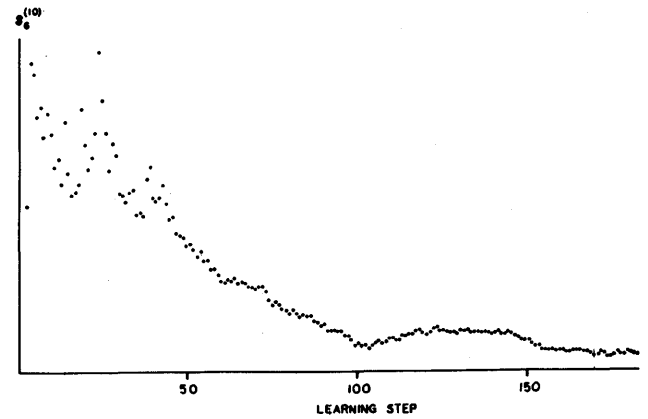


Figure 5—Convergence of the 10th component of s_6

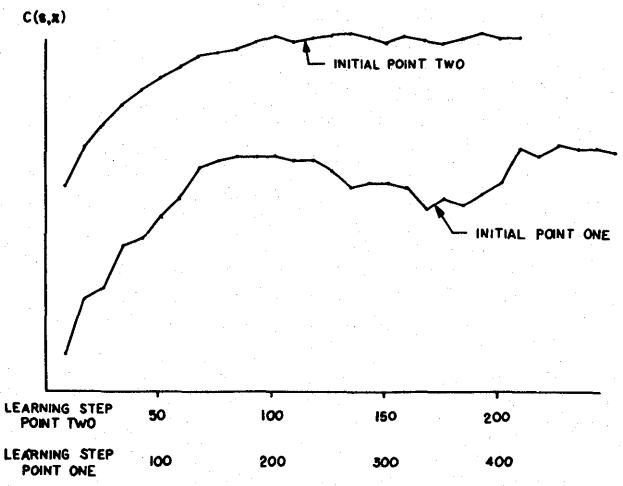


Figure 6—Improvement of $E\ C(s, x)$

channel for s_6 of the small data set. The computer listing for each step of learning was examined to find a rapidly changing component. This component is typical of the convergence of these values. Note that at the beginning there are rapid and random changes in its value due to the large value of a_n and the fact that the structure is rapidly changing. The learning then appears to enter a phase where the structure is rapidly descending toward a minimum. The last part of the learning seems to be random fluctuation about the optimum. Note that convergence appears to take only about 150 steps.

Local Optimums: It was found that there definitely was more than one optimum. By choosing different starting points, the algorithm converged to different optimums. To see this, examine Figure 6. This is a plot of the smoothed data for two runs with the small data set. Each learning run was made with the same data except that the starting points were different. It can be seen from the figure that the initial point one converged to a local optimum that was not as good as that for the initial point two. We can be fairly certain that the first point will never converge to the second, since more than twice the number of learning steps were run for point one than for point two.

The Standard Phonetic Structure: The output strings from sample words were inspected for similarities to standard phonetic spellings.[15] It was found that the two structures were similar in many respects. A one-to-one correspondence could be made between certain standard phonemes and machine phonemes. This was particularly true for consonants such as [*s*] or [*f*]. The two structures were not equivalent for vowels or glides with time changing spectra. In this case the machine structure appeared to develop phonemes that represented transitions between the standard phonemes.

RECOGNITION OF PHONEME STRINGS

In this section we present a means of classifying the phoneme strings that are produced as a result of sequential feature extraction. For completeness, we shall restate the recognition problem here. There is a set of symbols $A = \{a_1, \cdots, a_M\}$ called phonemes. The pattern space P is the set of all finite sequences of symbols from A. A typical pattern from P will be denoted either by the sequence $a_{\sigma_1}, \cdots, a_{\sigma_k}$ or by q. There are R pattern classes, each class has some characteristics associated with its sequences that differentiate it from the other classes.

If one were to use a Bayes decision procedure, the following would be needed. According to decision theory, in order to minimize the probability of error, the discriminant functions

$$g_i(q) = p(q \mid i)p(i), \quad i = 1, \cdots, R \qquad (16)$$

are needed, where $q \in P$, $p(q \mid i)$ is the probability of q given pattern class i, and $p(i)$ is the a priori probability of class i. If it can be assumed that the $p(i)$ are all equal, then they can be dropped from (16). The problem is then to estimate $p(q \mid i)$ for all q and i. It is obvious that even if the length of the strings is bounded, the estimation of all the probabilities in (16) is an almost impossible task for a phoneme set of any size. For example, if there are ten phonemes and the strings are assumed to be no longer than length 5, then the number of probabilities is greater than 5^{10}. The amount of data to estimate these probabilities is too large to obtain practically. Therefore, a Bayes decision procedure for this case is impractical. A decision procedure that does not require the estimation of all possible probabilities will have to be found.

In order to motivate the development that follows, we will outline the basic approach used for this recognition problem. A concept of the storage of prototype pattern strings is extended to what is called a *generalized prototype string*. This concept will be used in the pattern recognition problem as follows. The generalized prototype string will be defined as a truncated Markov chain. R of these Markov chains are defined. These Markov chains produce finite strings of symbols in P. The probability measures $p(q \mid i)$ on these strings are assumed to approximate the probability measures for each of the R pattern classes. These generalized prototype strings are used to define sequential machines for

recognition. These sequential machines will accept an unknown string q and calculate $p(q \mid i)$. This will then be used to classify q according to (16).

The need for generalized prototype strings comes from the fact that the intra-class variance of the strings is large. If this variance is small, then a straightforward method of recognition exists. This method would be to store the most common output strings for each class. Each unknown pattern string q would then be matched against the stored strings. If there is a match with one of the stored strings for class i, then q will be put in pattern class i. If, however, the strings within a pattern class show a large variance, too many strings will have to be stored in order to recognize a reasonable number of patterns. To reduce the storage requirements in this case, the following concept of a generalized prototype string has been formulated.

In order to simplify notation, we will work only with the indices of the strings and omit the symbols a. In other words, if we have a string $a_{\sigma_1}, \cdots, a_{\sigma_k}$, then we will describe this string as $\sigma_1, \cdots, \sigma_k$. This will cause no confusion.

If there are M phonemes in A, then the possible indices for the symbols a_i run from 1 to M. Assume that there is a new symbol a_{M+1} that represents string termination. That is, using the notation introduced above, each string is of the form $\sigma_1, \cdots, \sigma_k, M+1$. This will be useful when the truncated Markov chains are defined.

Suppose we have a prototype string $n_1, \cdots, n_m, n_{m+1} = M+1$. This string will be used to define a Markov chain that terminates when $M+1$ appears. To do this, assume that there exist probabilities $p(i)$, $i = 1, \cdots, m$, and $p(j \mid k)$, $j = k+1, \cdots, m+1$, $k = 2, \cdots, m+1$. These probabilities, along with the sequence defined above, can now be used to define a Markov chain. This chain will produce subsequences of $n_1, \cdots, n_{m+1}$. If $n_{i_1}, \cdots, n_{i_k}, n_{m+1}$ is such a subsequence, then, using the above probabilities, the Markov property allows us to write

$$p(n_{i_1}, \cdots, n_{i_k}, n_{m+1}) = p(i_1)p(m+1 \mid i_k)\prod_{j=2}^{k} p(i_j \mid i_{j-1}), \quad (17)$$

where $p(n_{i_1}, \cdots, n_{i_k}, n_{m+1})$ is the probability that this subsequence occurs, $p(i_1)$ is the probability that index i_1 is the first index in the subsequence, and $p(i_j \mid i_{j-1})$ is the probability that index i_j follows i_{j-1}. Note that if at any time $i_j = m+1$, the string terminates. Also note that the subsequence preserves the order of the original sequence. That is, $p(j \mid k) = 0$ for $j \leq k$.

In accordance with the above discussion we have the following definition.

Definition 4. A sequence $n_1, \cdots, n_m, n_{m+1} = M+1$ together with the probabilities $p(i)$, $p(j \mid i)$, $j = i+1, \cdots, m+1$, $i = 1, \cdots, m$ is called a *generalized prototype string* S. The string S is said to be *generated* by $n_1, \cdots, n_m, n_{m+1}$.

Definition 5. The *range* of a generalized prototype string S is that set of subsequences Q such that a subsequence $n_{i_1}, \cdots, n_{i_k}, n_{m+1}$ is in Q if and only if $p(n_{i_1}, \cdots, n_{i_k}, n_{m+1}) > 0$.

Thus, the generalized prototype string is actually a probability measure on P. Suppose that $\sigma_1, \cdots, \sigma_k, M+1$ is a string in P. If this sequence is not in the range of S, then there is a subsequence $n_{i_1}, \cdots, n_{i_k}, n_{m+1}$ such that $\sigma_j = n_{i_j}$ for all j. The probability measure on P is defined in the following manner.

Definition 6. If $\sigma_1, \cdots, \sigma_k$ is a sequence in P, then define

$$p(\sigma_1, \cdots, \sigma_k) = 0, \quad \text{if } \sigma_1, \cdots, \sigma_k \text{ is not in the range of } S.$$

$$= p(n_{i_1}, \cdots, n_{i_k}, n_{m+1}), \quad \text{if } \sigma_1, \cdots, \sigma_k \text{ is in the range of } S, \quad (18)$$

where $n_{i_1}, \cdots, n_{i_k}, n_{m+1}$ is such that $\sigma_j = n_{i_j}$ for all j, and $p(n_{i_1}, \cdots, n_{i_k}, n_{m+1})$ is defined in (17). The probability in (18) will be called *the measure associated with* S.

It can now be seen that the usual notion of a prototype string is a special case of the generalized prototype string. If $p(1) = 1$, and $p(i \mid i-1) = 1$ for all i, then the resulting range of S consists only of the string $n_1, \cdots, n_{m+1}$. This corresponds to the method described at the beginning of this section.

The following approach to the recognition problem will now be taken. Each pattern class i has a probability measure $p(q \mid i)$ associated with it. It is assumed that there exist generalized prototype strings S_i, $i = 1, \cdots, R$, such that $p(q \mid i)$ is the measure associated with S_i. In other words, we are using the concept of the generalized prototype string to approximate the measure $p(q \mid i)$. The learning procedure will require that each S_i be determined. A recognition procedure must also be developed that will allow each of the $p(q \mid i)$ to be evaluated for an unknown pattern. To simplify notation, write $p_i(q) = p(q \mid i)$.

The calculation of $p_i(q)$ associated with each S_i can be implemented by the use of sequential machines. It was shown that the measure on the range of S_i was the result of a truncated Markov chain. Let $n_1{}^i, \cdots, n_{m_1}{}^i, M+1$ be a string that generates S_i. Assume for the moment that this string and the associated prob-

abilities have been completely determined. The sequential machine that implements S_i contains $m_i + 2$ states. These states are labeled (0), 1, 2, $\cdots$, m_i, $m_i + 1$, where (0) is the reset or power on state and $m_i + 1$ is the terminal state. In other words, each machine state j is associated with the corresponding term $n_j{}^i$, for $j = 1, \cdots, m_i + 1$. The (0) state corresponds to the start of the sequence. Each state transition is defined in the following manner.

Let $p_i(j)$, $p_i(k \mid j)$, $k = j + 1, \cdots, m_i + 1$, $j = 1, \cdots, m_i$ be the probabilities used to define S_i. Suppose the sequential machine M_i is in state (0). If $p(j) \neq 0$ define a transition to state j for input symbol n_j. If $p(k) = 0$ for some k, there is no transition from (0) to state k. Now suppose the sequential machine is in state k. If $p(j \mid k) \neq 0$, define a state transition from state k to state j for input n_j. Continue for all such states from 1 to m_i. This process completely defines the sequential machines M_i.

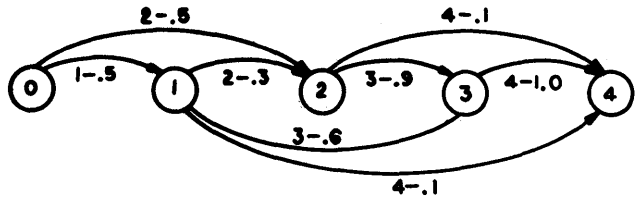
Definition 7. The sequential machine M_i is said to *accept* a sequence $\sigma_1, \cdots, \sigma_k$ if this sequence is contained in the range of S_i.

We are now in a position to see how the above sequential machines M_i can calculate the probabilities $p_i(q)$. Recall that each state transition was defined using one of the probabilities $p_i(j)$, or $p_i(j \mid k)$. Thus, each state transition has an associated probability. If machine M_i accepts pattern string q, then there is a sequence of state transitions leading to the final state $m_i + 1$. Each state transition has an associated probability. If the product of all these probabilities is formed, then it is seen that the result is the product in (17). But, it has been seen that this is the desired probability $p_i(q)$ from Definition 6.

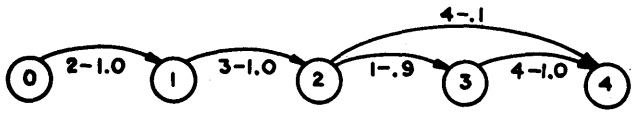
Therefore, the recognition procedure is as follows. The unknown string q is applied to all the sequential machines M_i, $i = 1, \cdots, R$. If none of the machines accept q, then it is rejected as unrecognizable. For each machine M_i that accepts q the probability $p_i(q)$ is calculated in the following manner. An accumulator register is initialized to the value 1 at the start of q. As each state transition is made during the application of q the probability associated with that transition is multiplied by the contents of the accumulator, and the result is stored back in the accumulator. After the machine reaches the final state $m_i + 1$, the desired probability $p_i(q)$ is in the accumulator. These calculated probabilities are then used to classify q in the sense of (16).

Two examples will now be given that will clarify the above development.

Example 2. Suppose there are three phonemes in A.



(a)



(b)

Figure 7—(a) State transition diagram for pattern class 1 with associated probabilities, and (b) State transition diagram for pattern class 2 with associated probabilities

That is, $A = \{1, 2, 3\}$. Assume there are two pattern classes and that the prototype strings for these two classes are

1, 2, 3, 4

2, 3, 1, 4

The transition probabilities are as follows:

$p_1(1) = .5$	$p_1(1 \mid 2) = .3$	$p_1(2 \mid 3) = .9$
$p_1(2) = .5$	$p_1(1 \mid 3) = .6$	$p_1(2 \mid 4) = .1$
$p_1(3) = 0.0$	$p_1(1 \mid 4) = .1$	$p_1(3 \mid 4) = 1\,0$
$p_2(1) = 1.0$	$p_2(1 \mid 2) = 1.0$	$p_2(2 \mid 3) = .5$
$p_2(2) = 0.0$	$p_2(1 \mid 3) = 0.0$	$p_2(2 \mid 4) = .5$
$p_2(3) = 0.0$	$p_2(1 \mid 4) = 0.0$	$p_2(3 \mid 4) = 1.0$

These sequences and probabilities can be used to design the sequential machines shown in the state transition diagrams in Fig. 7. By inspection of the state transition diagrams it can be seen that the range for S_1 consists

of the strings

1234
124
134
14
24
234

The measure associated with S_1 is then seen to be

$$p_1(1234) = .5 \times .3 \times .9 \times 1.0 = .135$$

$$p_1(124) = .5 \times .3 \times .1 = .015$$

$$p_1(134) = .5 \times .6 \times 1.0 = .3$$

$$p_1(14) = .5 \times .1 = .05$$

$$p_1(24) = .5 \times .1 = .05$$

$$p_1(234) = .5 \times .9 \times 1.0 = .45$$

In the same manner, the range for S_2 is

234
2314

The measure associated with S_2 is

$$p_2(234) = 1.0 \times 1.0 \times .1 = .1$$

$$p_2(2314) = 1.0 \times 1.0 \times .9 \times 1.0 = .9$$

Note that the ranges for S_1 and S_2 overlap in that 234 is common to both. But 234 will be put in class 1 since $p_1(234) > p_2(234)$.

There is a subtle point about the application of the Markov chain. While the strings in the range of a generalized prototype string are assumed to be produced as a result of a Markov process, the strings themselves are not Markov. The next example will illustrate this point.

Example 3. Consider the generalized prototype string shown in Fig. 8 (Sequential machines will be used to define the S_i from this point on since the notation is more compact). The range of this generalized prototype string is

1324
134
234
2324

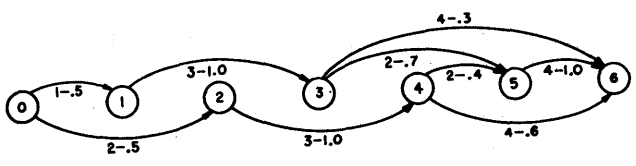


Figure 8—Example of prototype string whose range is not Markov

(here, A is the same as in Example 2). If these strings were produced by a Markov process, then $p(4 \mid 13) = p(4 \mid 23)$ and $p(2 \mid 13) = p(2 \mid 23)$. But, from the figure it can be seen that this is not the case. For $p(4 \mid 13) = .3$, and $p(4 \mid 23) = .6$. Also, $p(2 \mid 13) = .7$ and $p(2 \mid 23) = .4$. Thus, while the range of S is produced by a Markov process of the states of the sequential machine, the resulting strings in the range are not Markov.

TABLE I—Example of Table for Sequential Structure

original strings

1	5	1	2	4	3	
1	2	3				
1	2	3				
1	2	3				
1	2	3				
6	5	1	2	3		
1	2	3				
1	2	3				
5	1	2	6	3		
1	2	4	3			
1	2	4	3			
1	2	3				
1	5	1	2	3		
1	2					
5	1	2	4	3		
6	5	2	3			
1	2					
1	2	3				
1	2	6	3			
5	1	2	3			
5	6	2	6	3		
1	5	6	3			
5	1	2	4	3		

1	5	1	2	4		3
1			2			3
1			2			3
1			2			3
1			2			3
6	5	1	2			3
1			2			3
1			2			3
5		1	2		6	3
1			2	4		3
1			2	4		3
1			2			3
1	5	1	2			3
1			2			
	5	1	2	4		3
6	5		2			3
1			2			
1			2			3
1			2		6	3
	5	1	2			3
	5	6	2		6	3
1	5	6				3
	5	1	2	4		3

Unfortunately, the method for learning the S_i is not as formal as the preceding. The prototype strings were determined from the tables of sample strings. For example, consider the sample strings in Table I. Each string has been listed in the table so that each column contains only one phoneme. The strings have been arranged so that the order of the phonemes is unchanged and the number of columns is minimized. These tables were formed by a manual procedure, and at present this procedure cannot be written as a sequence of steps. Once these tables have been determined, the prototype sequences can be defined. Each of the columns in the tables contains a distinct phoneme. These phonemes are taken to be the prototype string that generates S_i for each class i. This is best described through the use of an example. Consider the table for "one" in Table II. For this case

$$A = \{1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22\}$$

The prototype string that generates S_1 for the class "one" is 10, 22, 20, 17, 16, 11, 17, 11, 1, 15, 5, 3, 1, 15, 3. Crude estimates of the probabilities can also be made by counting the transitions between states as defined by rows in the table. Using these probabilities and the above prototype string, we have the generalized prototype string for "one" in Figure 9.

TABLE II—Samples of Output Strings for "One"

10		20	17		11		1	15		1		5	1	15
	22				11	17	1							
10			17		11		1	15						
10			17					15		1	15		1	
10		20	17		11		1							
	22		17		11		1	15		1				
				16	11		1	15		1				
10	22				11		1							
	22				11		1	15		1				
10	22				11			15		1				
		20	17		11			15						
	22				11		1							
10			17		11	17	1	15						
10	22				11	17		15						
10		20	17		11		1	15						
	22		17		11		1	15						
10		20	17		11		1	15				5	1	15
10		20	17					15		1				
		20	17					15		1				
	22		17		11			15		1				
			17		11	17		15		1		5	1	15
10			17		11			15						
10		20	17		11		1	15						
	22	20	17		11		1	15	3			5	3	15 3
	22				11		1	15		1				

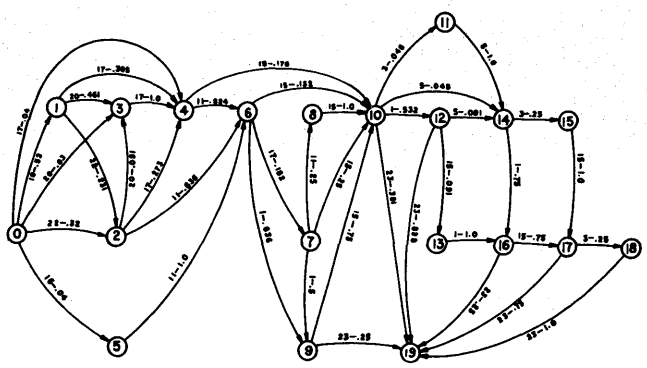


Figure 9—State transition diagram for "ONE"

A computer program was written that simulated the entire recognition system. The program accepted isolated words, computed the phoneme strings, and implemented the sequential machines. The sequences in the training set were recognized using the sequential machines. Using this method, the recognition rate was one hundred percent for the 250 patterns in the training set.

In order to further demonstrate the power of the algorithm, the recognition was run using a restricted system. The association of probabilities with state transitions was removed. Recall that the calculated probabilities for an imput pattern string were used only if more than one machine accepted the string. In order to provide a more severe test for the concept of using sequential machines, the use of the calculated probabilities was dispensed with. In this case, only one pattern from the 250 was accepted by more than one machine. One sample pattern for "three" was accepted by both the machine for "three" and "two". Thus, under these circumstances, the sequential machines performed most satisfactorily.

It was desirable to continue to simplify the algorithm in order to see when the performance began to degenerate. Therefore, the following additional simplification was tried. The assumption that each input string had phoneme $M + 1$ in the terminal position was dropped. Using the same definition of acceptance, this implies that a sequential machine will accept an input string even if the machine is not driven to its final state. It is stressed here that these changes in the algorithm were not made to try to improve performance, but were made to try to see how far the algorithm could be degraded and still achieve good results.

Recognition was attempted using the strings without the terminal symbol described above. The sequential

machines with no probabilities were used. If more than one machine accepted a pattern, then it was assumed to be an error. Under these conditions, the error rate was 4%. That is, there were ten error patterns out of the 250 presented to the system. The confusion matrix for this test is given by

		decision class									
		1	2	3	4	5	6	7	8	9	0
pattern class	1	25									
	2		22		1				1		
	3		1	20						2	
	4				24						
	5					24					
	6						25				
	7						3	21			
	8								25		
	9		1							23	
	0										25

This algorithm represents considerable simplification of the full algorithm. Under these conditions the error rate was still low. We can conclude that the potential of the complete algorithm is such that further work is highly desirable.

The conclusions that can be made from this section are as follows. It has been seen that sequential feature extraction has considerable utility for use in the two stage recognition procedure presented here. The structure produces phoneme strings that can easily be used to design the sequential machine for the second stage of recognition. The recognition results in the second stage were encouraging. The initial motivation for development of the second stage recognition was to demonstrate the capabilities of the machine phonetic structure. However, the experimental results indicate that this has potential for solving recognition problems independent of the sequential feature extraction.

CONCLUSIONS

In this study we have presented a means of detecting sequential structures in waveforms for recognition. This process is called sequential feature extraction. The waveforms were assumed to be produced by a random process that was unknown. A learning algorithm that automatically generated a structure for sequential feature extraction was presented. This learning rule was unsupervised, and was shown to be a generalization of previously unsupervised learning rules for the time invariant case.

It was shown that sequential feature extraction could be considered as a transformation T_s from W to the set of all finite sequences of symbols from a set A called the phoneme set. A structure on T_s was developed so that the transformation was dependent on a set of parameters $\mathbf{s}$. This allowed us to find an $\mathbf{s}$ that was optimum with respect to a performance function.

This algorithm was applied to a problem in speech recognition. Experimental results were given that showed interesting relationships between the standard phonetic structure and the structure developed by sequential feature extraction. It was concluded that the automatically developed structure was related to the linguistic structure, but that there were significant differences due to the continuously time changing character of speech.

A new concept called the generalized prototype string was presented. This was a generalization to the probabilistic case of the method of storage of prototype strings. Each generalized prototype string was seen to be a means of approximating the probability measures on P. Once the generalized prototype strings were found, it was possible to design sequential machines for recognition. These sequential machines were seen to implement the calculation of estimates of the individual pattern class probabilities $p_i(q)$, where $q \in P$. Using these probabilities, the pattern could be classified according to Bayes decision theory.

The sequential feature extraction presented in this study represents a new approach to waveform recognition and unsupervised learning. For the case of speech, it was seen that sequential feature extraction was related to a phonetic structure. While phonetic structures are not new, the concept of using unsupervised learning to automatically develop a phonetic structure is new. The sequential structure in speech or other types of waveforms can be detected by using this algorithm. Because the algorithm is automatic, there is no bias due to previous results from linguistics. This is a particular advantage if the algorithm is to be applied to applications other than speech.

The restriction of the unsupervised learning algorithm to the time invariant case showed that the algorithm had advantages over current methods. No knowledge of the probability measures is required, strict separa-

bility of clusters is not required, the class of allowed metrics is large, and there is no requirement that the sample patterns be stored for processing since the algorithm will accept patterns one at a time.

REFERENCES

1 B P LATHI
Signals, systems and communication
Wiley New York 1965

2 S C FRALICK
Learning to recognize patterns without a teacher
IEEE Trans Inf Th Vol 13 pp 57–64 January 1964

3 D P COOPER P W COOPER
Adaptive pattern recognition without supervision
Proc IEEE International Convention 1964

4 E A PATRICK J C HANCOCK
Nonsupervised sequential classification and recognition of patterns
IEEE Trans on Inf Th Vol 12 July 1966

5 C G HILBORN D G LAINIOLIS
Optimal unsupervised learning multicategory dependent hypothesis pattern recognition
IEEE Trans Inf Th Vol 14 May 1968

6 E M BRAVERMAN
The method of potential functions in the problem of training machines to recognize patterns without a teacher
Automation and Remote Control Vol 27 October 1966

7 G H BALL D J HALL
ISODATA—An iterative method of multivariate analysis and pattern classification
International Communications Conference Philadelphia Pennsylvania June 1966

8 J GAZDAG
A method of decoding speech
University of Illinois AD 641 132 June 1966

9 K W OTTEN
Simulation and evaluation of phonetic speech recognition techniques—Vol III Acoustical characteristics of speech sounds systematically arranged in the form of tables
NCR Company AD 601 422 March 1964

10 I LEHISTE
Acoustical characteristics of selected English consonants
Int J Am Linguistics Vol 30 July 1964

11 W F MEEKER A L NELSON P B SCOTT
Voice to teletype code converter research program. Part II—Experimental verification of a method to recognize phonetic sounds
Technical Report ASD-TR61-666 Part II AD 288 099 September 1962

12 K W OTTEN
Simulation and evalutation of phonetic speech recognition techniques Vol II—Segmentation of continuous speech into phonemes
NCR Company AD 601 423 March 1964

13 J L FLANNAGAN
Speech analysis, synthesis and perception
Springer-Verlag New York 1965

14 M KABRISKY
A proposed model for visual information processing in the human brain
University of Illinois Press Urbana Illinois 1966

15 J S KENYON T A KNOTT
A prouncing dictionary of American English
G & C Merriam Springfield Massachusetts 1953

Pulse-Amplitude Transmission System (PATSY)*

by NEAL L. WALTERS

IBM Corporation
Research Triangle Park, North Carolina

SUMMARY

A new type of pulse-amplitude transmission system (PATSY) has been developed to transmit numeric data asynchronously over relatively short distances. The "transmitting station" requires no power or data set. As many as 32 transmitting stations can be attached to a single "receiving station." Other advantages are simplicity, flexibility, and low cost.

The low-speed system has similarities to a Touch-Tone** telephone, using resistors instead of tone oscillators. Resistors are less expensive and may be more reliable. The adapter in the sending station operates similarly to an ohmmeter which reads resistances in either polarity.

INTRODUCTION

A new type of pulse-amplitude transmission has been developed to transmit numeric data asynchronously at low rates* over short distances (up to 1,000 feet). The transmission is unique in that the terminating "data entry unit" requires no power or data set. Other advantages are two-wire lines which are easy to install and change, small amount and simplicity of hardware, and low cost.

The pulse-amplitude transmission system (PATSY) is being used to transmit data on personnel and machine performance, materials, job status, etc., between data entry units and area stations. Up to 32 data entry units can be attached to one area station.

The data entry units are polled (scanned) by the area stations, at a rate of 250 units a second, to determine when one is ready to send data. Line turnaround is relatively unimportant because lines are short and data flow is primarily one-way (from the terminating data entry units into the area stations). The only transmissions out to the data entry units are a signal to start the unit reading when the scanner in an area station finds a data entry unit requiring service, and an acknowledgment of a data entry indicating whether it has been satisfactorily received.

The low-speed system has similarities to a Touch-Tone telephone except that resistors have been substituted for the tone oscillators (Figure 2). Resistor-diode circuits are less expensive and may be more reliable than oscillator circuits. They are arranged, with two diodes, into two banks. By sampling with alternate positive and negative pulses, one particular resistance value (and, thus, character) out of the character set can be identified. Alternate pulsing makes it possible to use half as many resistors with twice the separation between values, and thus obtain more reliable identification.

Therefore, the operation of the PATSY adapter in the area station is similar to that of an ohmmeter, capable of reading resistances in either polarity. When a data entry unit is ready, it informs the area station by placing a short circuit on its pair of wires.

The area station then sends a voltage pulse as a start command to the data entry units, which begins placing combinations of resistances on the wires, interspersed with "open-circuit" periods. The area station uses the open-circuit condition to provide timing information between characters, and reads the resistance levels to determine what characters are being transmitted. When the transmission has been completed, the area station sends another voltage pulse to the data entry unit to indicate that the message has been received.

---

* PATSY is not a pulse-amplitude transmission system in the sense that the transmitting station sends pulses to the receiving station. However, the data appearing on the transmission lines takes the form of amplitude-encoded pulses.

** Trademark of Bell Telephone Company.

* At 40 cps in the IBM 2790 Communication System for which developed (Figure 1), but could be used for higher rates.

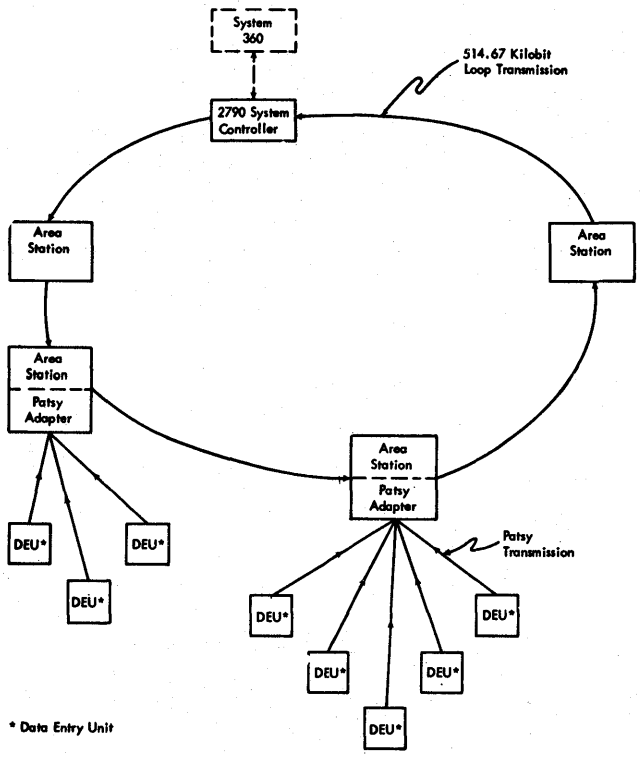


Figure 1—IBM 2790 communication system

THE PATSY INTERFACE

To distinguish between the two wires connecting the Data Entry Unit and the PATSY adapter, one wire will be called the "high line" and the other the "low line." Values of resistance placed across the lines by the Data Entry Unit will be referred to as follows: resistances measured with the high line positive with respect to the low line will be labeled "polarity 1" (P_1) and resistances measured with opposite polarity will be labeled "polarity 2" (P_2). There are five resistance levels in the PATSY code. They are: R_0, an open circuit; R_1, R_2, and R_3 in order of descending resistance; and R_4, a short circuit. Transmission over

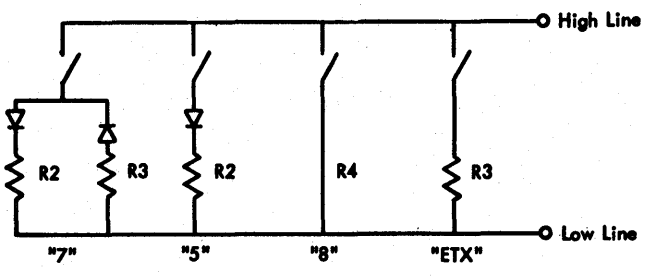


Figure 2—Examples of PATSY characters

PATSY occurs in the sequence shown in the following paragraphs.

Inactive (used for diagnostic testing)

In the inactive state, a data entry unit will have R_3 with polarity 2 across the lines. The polarity 1 resistor is not used but the polarity 2 resistance allows the system to check the wiring and connections to the unit in a diagnostic-test mode by having the adapter read that P_2 resistance. In this way, shorted or open lines and missing units can be detected.

Request for service

When a data entry unit is ready to send data, it makes a request for service by a R_4, P_1. This resistor must remain on the line until sometime after the start command. The P_2 impedance is not read.

Start command

The Start Command sent by the PATSY adapter to the data entry unit is a P_2 voltage pulse; that is, the low-line is positive with respect to the high line. Pulse amplitude is 30 volts with a duration of approximately 60 milliseconds, and it is able to drive a 100-ohm load.

Data transmission

The data entry unit is required to provide a P_1 open circuit continuously for a minimum of three milliseconds preceding each character. For the first character, this open circuit time begins after the start command pulse has finished. There is no limit on how long an open circuit can last (within the constraints of the time-out criteria of the particular program controlling the system).

After the open-circuit time, the data entry unit places the P_1 resistance on the line (which will be in the normal high-line positive state for open-circuit time and the first part of the character reading). After approximately five milliseconds, the polarity on the line will be reversed by the adapter and the P_2 resistance will be read. After six more milliseconds, the line will be reversed to wait for the next P_1, open circuit condition.

These two resistances, of opposite polarity, constitute a character. They may be placed across the line at the same time and left across the line for a minimum of 12 milliseconds. The maximum time that they can be left on the line is only limited by system time-out constraints. Opening the circuit by removing the P_1 re-

1st Polarity Reading \ 2nd Polarity Reading	R_0	R_1	R_2	R_3	R_4
R_0	Undetectable				
R_1	*	*	=	*	*
R_2	5	*	4	7	6
R_3	1	-	Space	ETX	2
R_4	9	*	0	3	8

* Unassigned

TABLE I—2790 5-Level PATSY Code

sistor indicates to the receiving station that character transmission has been completed.

Table I shows the coding used for PATSY characters. The rows represent the P_1 half of a character, the columns the P_2 half. R_0 is illegal as the P_1 half of a character since a P_1, open circuit is reserved for the timing information used to separate characters. The 10 numbers, dash, equal sign, space and ETX (end of text) are used, leaving six unassigned characters.

End request

The particular combination of resistances, R_3, with polarities 1 and 2, constitute the end request from a data entry unit, and is sent as the last data character. It is not necessary that the end request be followed by an open circuit. When the PATSY ETX character is detected, reading ceases and the receiving station waits for instructions from the system controller on how to answer the data entry unit.

End command

After the PATSY data entry unit makes an end request, the system will send an end command. The command will be either a normal end command indicating the message has been received with no detectable errors, or an error end meaning an error has been detected. In addition, the system will send an error end command a predetermined time after the start command if no end request is detected. This time-out will depend on the type of data entry unit.

A normal end command will return the data entry unit to its inactive state and is the same as the start command except that it lasts twice as long. An error end command has the same amplitude and duration of the normal end command but will have the opposite polarity. The error end command inhibits further operation of the data entry unit until the operator resets the terminal. He is informed of the error by a red button popping up.

RESISTANCE READING

Resistance values are sensed by the circuit shown in Figure 3. The plus voltage is used as the transmission voltage as well as the circuit supply. In the block diagram (Figure 3a), the reference resistors of block 3 provide the high and low parts of a voltage divider with the data-entry-unit resistor, plus wire and other unknown impedances. The matched 200-ohm reference resistors were chosen to minimize noise pickup by their relatively low impedance and the desire to keep the line balanced. The polarity switch and low-pass filter are omitted from Figure 3b for simplicity.

This combination of impedances produces a difference voltage, V_d, which is the input to a differential amplifier,

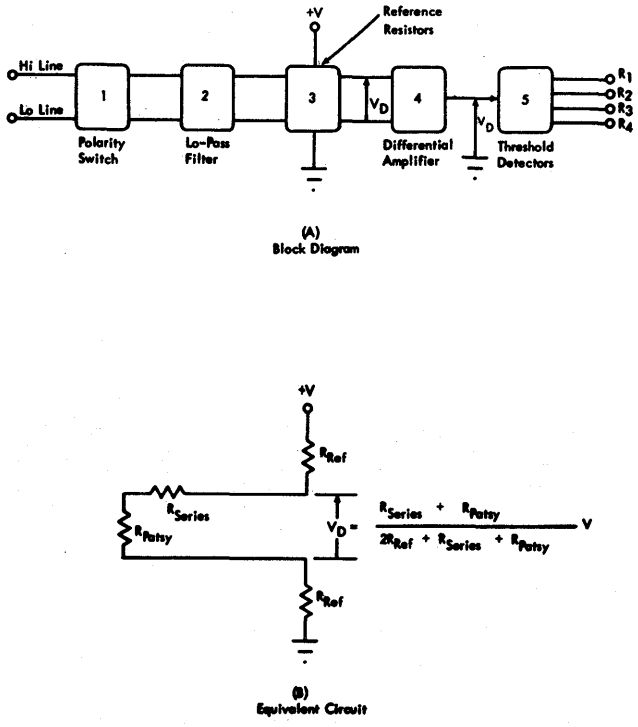


Figure 3—PATSY reading circuit

TABLE II—Character Timing

Time (milliseconds)	Condition
—	Open circuit
0	Open circu condition ceases
0–1	Low-pass filter is allowed to settle
1–4	Positive half is read* and stored
4	Polarity is reversed
4–7	Low-pass filter is allowed to settle
7–10	Negative half is read* and stored
10	Polarity is reversed
10½	Character is transmitted to system controller
11	Timing holds for open circuit
$11 + t, 0 < t$	Open circuit, timing is reset

* For these three milliseconds, the value of the lowest threshold detector which is crossed is stored. For example, δ_1 may be crossed, then δ_2, and finally δ_3 is crossed for only a few microseconds in the three millisecond period. The reading will be taken as R_3.

which has a gain of one, and is used for common-mode rejection. At the output of the amplifier, threshold detectors are used to determine when V_d drops to specific levels. Threshold detector one indicates when V_d is less than the level $E_{cc} - \delta_1$; threshold detector two indicates when V_d is less than $E_{cc} - \delta_2$ ($\delta_2 > \delta_1$), etc. The value of the differential voltages δ_1, δ_2, etc., are determined by the transmission-resistor values. (See Appendix.) The number of threshold detectors are the same as the number of discrete resistor values to be read, and hence, determine the size of the character set. In addition, threshold detector one indicates the open-circuit condition since any resistance too large to satisfy the first threshold is defined as an open circuit.

On the data-entry-unit side of the reference impedances is a polarity switch and a low-pass filter which rolls off at 600 Hz or about ten times greater than the maximum-character rate. The purpose of the switch is to read the data-entry-unit resistance value in both directions. In block one of Figure 3a, normally point a is connected to b, and c to d, but the connection can be reversed where point c is connected to b, and a to d, all under control of the logic. Point b on the polarity switch is always positive with respect to point d, which simplifies the low-pass filter and difference amplifier.

As stated previously, an uninterrupted period when there is a positive open circuit must precede each character. When this open circuit is ended, a reading sequence begins under logic timing control. Table II shows the steps in reading a character.

Figure 4 shows the waveforms on the high and low lines for a complete message transmission from a data entry unit to a PATSY adapter.

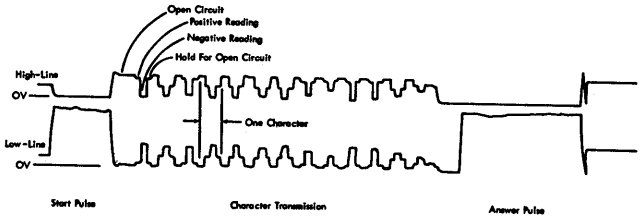


Figure 4—High-and low-line PATSY waveforms

POLLING AND LINE MULTIPLEXING OF DATA ENTRY UNITS

The polling and line configuration is shown in Figure 5. Up to 32 data entry units can be attached to a single PATSY adapter. There are two groups of multiplex switches, the high-line and low-line switches.

For polling (scanning) data entry units, all the high-line switches are closed and all the low-line switches are open. The resistance-sensing circuit is used in an un-

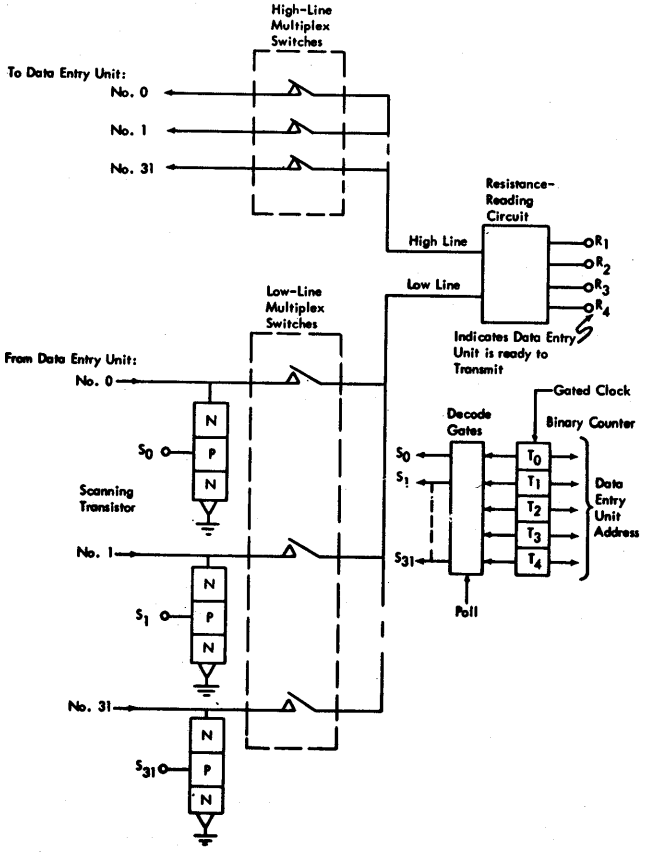


Figure 5—Polling and addressing of data entry unit

balanced mode since there is no connection to the low-line input. Each low line has a scanning transistor. When one of these transistors is turned on, a current path exists from the high line of the resistance-reading circuit through the particular data entry unit being polled, and to ground through the scanning transistor. When a data entry unit desires service, it places a P_1 short circuit on its wires. When that data entry unit is polled, the voltage on the high line of the resistance reading circuit is pulled down toward ground, making V_d equal approximately 0, satisfying the lowest threshold and indicating a request.

Polling is under the control of a binary counter. When polling, the counter is advanced by clock pulses every four milliseconds. A counter decode is used to sequentially turn on the scanning transistors, 250 data entry units being scanned each second. When a request for service is detected, the clock pulses are degated from the counter for the duration of the time that that data entry unit is being read. Thus, the binary number in the counter becomes the data entry unit address, and the transmission of an address by the data entry unit is unnecessary. The identification of the transmitting data entry unit by the pair of wires to which attached is a novel feature in PATSY.

When a request is detected, polling ceases and the scanning transistors are no longer used. However, the decode now is used to close the proper high-line and low-line switches between the data entry unit and the receiving station to provide a dedicated connection.

Figure 6 shows the location of the 30-volt switch circuits which provide the Start Command to initiate a transmission and the End Command to terminate a transmission. The resistance reading circuit is disconnected from the lines when the 30 volts is applied. Identical circuits attached to both the high and low lines allow the 30 volts of either polarity to be applied to the data entry unit.

TRANSMITTING STATION

PATSY permits the attachment of a wide variety of data entry units to an adapter in a data collection system. The data entry units can be either mechanical, using brush commutators, or employ solid-state technology such as silicon-controlled rectifiers (SCR's). There is enough power in the start and answer pulses to pick relays or energize solenoids. In addition, there is no restriction on how slowly the data can be transmitted, permitting live keyboards to be intermixed with fixed-rate devices on the same adapter. In addition, the receiving station is transparent to message length. The first character transmitted is a transaction code, telling the system what type of message will be transmitted. The second character tells the system what type of data entry unit is sending, indicating the message length to expect.

Figure 7 is a schematic of a mechanical data entry unit used in the IBM 2790 Communication System. A brush commutator is on the left; a card/badge reader in matrix form on the right. When a card or badge is to be read, there is one electrical connection made on each row. The commutator is cocked in the request position. When the receiving station sends a start command, the magnet is picked releasing the commutator to scan through the rows, alternately placing the character resistors and an open circuit across the lines. The commutator is spring powered. Note the simplicity of the transmission components required to transmit the 12-character set of this station: six resistors and 15 diodes.

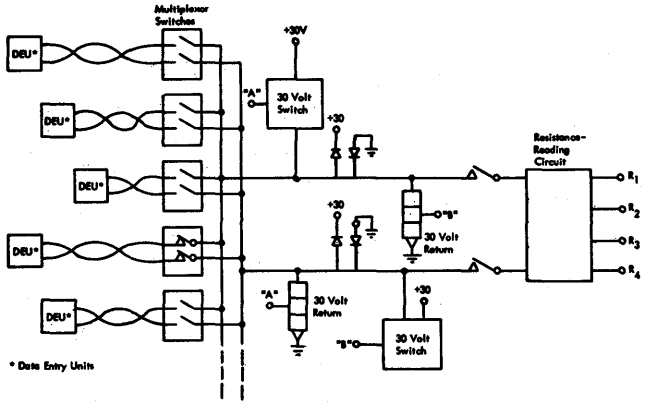


Figure 6—30-volt start and answer circuits

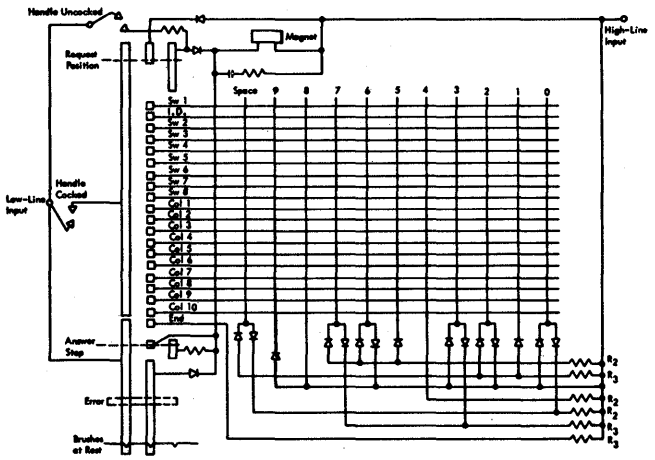


Figure 7—Data entry unit schematic

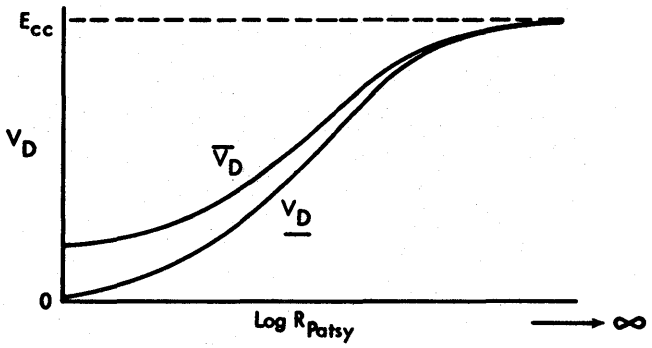


Figure 8—Voltage band for V_D

APPENDIX—SELECTION OF PATSY TRANSMISSION RESISTOR VALUES

Neglecting the effects of the polarity switch, the circuit used to read PATSY transmission is shown in Figure 3b. Where $R_{REF} = 200$ ohms, R_{PATSY} is the transmission resistor value and R_{SERIES} is all other resistance, i.e., wire resistance, diode drop, etc. Worst case V_D maximum and minimum can be expressed as:

$$\bar{V}_D = \frac{\bar{R}_{SERIES} + \bar{R}_{PATSY}}{2R_{REF} + \bar{R}_{SERIES} + \bar{R}_{PATSY}} \overline{(+E_{cc})} \qquad (1)$$

$$\underline{V}_D = \frac{\underline{R}_{PATSY}}{2\bar{R}_{REF} + \underline{R}_{PATSY}} \underline{(+E_{cc})} \qquad (2)$$

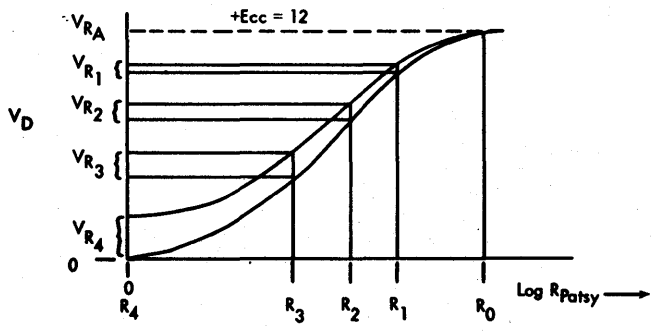


Figure 9—Location of PATSY resistors in voltage band

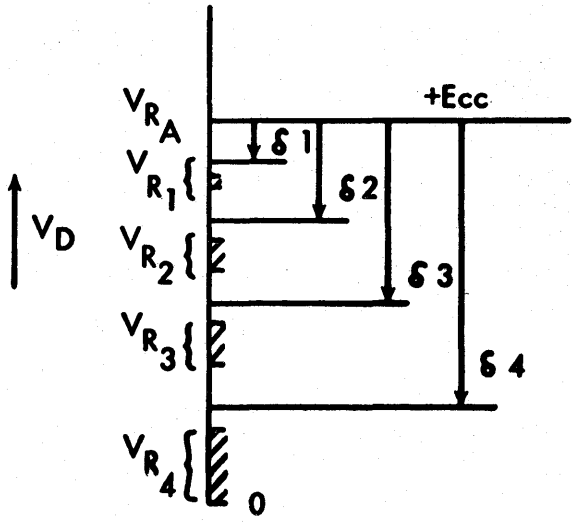


Figure 10—Placement of threshold levels

Where underlined values are minimums and overlined are maximums. The plot of $\bar{V}_D$ and $\underline{V}_D$ versus R_{PATSY} yields the curves shown in Figure 8.

The PATSY transmission resistor values were chosen to give the largest difference between $\underline{V}_D$ associated with one resistor value and $\bar{V}_D$ associated with the next smallest resistor value. The plot in Figure 9 shows the PATSY values on the V_D plot.

The voltage bands V_{R_1}, V_{R_2}, etc., are separated by guard-band voltages of approximately 1.5 volts.

When one of the PATSY resistors is being read, the difference voltage V_D will fall into one of the voltage bands V_{Rn}. The threshold voltages δ of the resistance reading circuit are then placed in the guard bands as shown in Figure 10.

The threshold voltages are placed *below* center of the guard bands because of the nature of the reading. When a resistance is being read, V_D is sampled for a period of three milliseconds, and the *lowest* threshold which is crossed in that time is considered to be the reading. Thus, the thresholds are offset to allow more noise rejection.

Also, it can be seen from the above diagram that four thresholds can be used to detect five levels where δ_1 indicates R_1 or lower if crossed and R_A if not crossed.

Termination of programs represented as interpreted graphs*

by ZOHAR MANNA**

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

This work is concerned with the termination problem of interpreted graphs. An *interpreted graph* can be considered as an abstract model of algorithms; it consists of a directed graph, where:

1. With each vertex v, there is associated a domain D_v, and
2. With each arc a leading from vertex v to vertex v', there is associated a total test predicate $P_a(D_v \rightarrow \{T, F\})$ and a total function $f_a(D_v^* \rightarrow D_{v'})$, where $D_v^* = \{x \mid x \in D_v \wedge P_a(x) = T\}$.

Let us represent by a state vector x the current values of the variables during an execution of an interpreted graph IG. An execution of IG may start from any vertex v with any initial state vector $x_0 \in D_v$. If during execution we reach vertex v with state vector x, $P_a(x)$ represents the condition that arc a (leading from v) may be entered, and f_a represents the operation of changing the state vector x to $f_a(x)$ when control moves along arc a. Execution will halt on vertex v, with state vector x, if and only if no predicate on any arc leading from v is true for x. An interpreted graph terminates if and only if all the executions of IG terminate.

Our main result is a sufficient condition for the termination of interpreted graphs defined by means of well-ordered sets. This result has applications in proving the termination of various classes of algorithms. Floyd[1] has discussed the use of well-ordered sets for proving the termination of programs.

* This work is based on the author's Ph.D. Thesis.[2] The work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

** Present address—Computer Science Department, Stanford University, Stanford, California.

WELL-ORDERED SETS

A pair $(S, \succ)$ is called an *ordered set*, provided that S is a set and $\succ$ is a relation defined for every pair of distinct elements a and b of S, and satisfies the following two conditions:

1. If $a \neq b$, then either $a \succ b$ or $b \succ a$;
2. If $a \succ b$ and $b \succ c$, then $a \succ c$ (i.e., the relation is transitive).

A *well-ordered set* W is an ordered set $(S, \succ)$ in which every non-empty subset has a first element; equivalently, in which every decreasing sequence of elements $a \succ b \succ c \cdots$ has only finitely many elements. For example,

1. I_1^+ (the set of all non-negative integers) is well-ordered by its natural order, i.e., $\{0, 1, 2, 3, \cdots\}$.
2. I_n^+ (the set of all n-tuples of non-negative integers for some fixed n, $n \geq 1$) is well-ordered by the usual lexicographic order, i.e., $(a_1, a_2, \cdots, a_n) \succ (b_1, b_2, \cdots, b_n)$ if and only if $a_1 = b_1$, $a_2 = b_2$, $\cdots$, $a_{k-1} = b_{k-1}$, $a_k > b_k$ for some k, $1 \leq k \leq n$.

DIRECTED GRAPHS

A *directed graph* G (*graph*, for short) is an ordered triple $\langle V, L, A \rangle$ where:

1. V is a non-empty finite set of elements called the *vertices* of G;
2. L is a non-empty set of elements called the *labels* of G; and
3. A is a non-empty set of ordered triples $(v, l, v') \in V \times L \times V$ called the *arcs* of G.

Note that L and A may be infinite sets. If L and A are finite sets, G is called a *finite directed graph*.

A finite path of a graph G (*path*, for short) is a finite sequence of n, $n \geq 1$, arcs of G of the form:

$$(v_{i_1}, l_{i_1}, v_{i_2}), (v_{i_2}, l_{i_2}, v_{i_3}), (v_{i_3}, l_{i_3}, v_{i_4}), \cdots, (v_{i_n}, l_{i_n}, v_{i_{n+1}})$$

[notation: $v_{i_1} \xrightarrow{l_{i_1}} v_{i_2} \xrightarrow{l_{i_2}} v_{i_3} \xrightarrow{l_{i_3}} v_{i_4} \cdots v_{i_n} \xrightarrow{l_{i_n}} v_{i_{n+1}}$].

We say that:

1. The path *joins* the vertices v_{i_1} and $v_{i_{n+1}}$ and *meets* the vertices $v_{i_1}, v_{i_2}, \cdots, v_{i_{n+1}}$.
2. The path is *elementary* if the vertices $v_{i_1}, v_{i_2}, \cdots, v_{i_{n+1}}$ are distinct.
3. The path is a *cycle* if the vertex v_{i_1} coincides with the vertex $v_{i_{n+1}}$; it is an *elementary cycle* if in addition the vertices $v_{i_1}, v_{i_2}, \cdots, v_{i_n}$ are distinct.

We define a *cut set* of a graph G as a set of vertices having the property that every cycle of G meets at least one vertex of the set.

A graph G is said to be *strongly connected* if there is a path joining any ordered pair of distinct vertices of G.

Let G be a graph $\langle V, L, A \rangle$. We define a *subgraph* $G_1 = \langle V_1, L, A_1 \rangle$ of G as the triple consisting of V_1, L and A_1, where V_1 is a subset of V and A_1 is defined by $A_1 = A \cap (V_1 \times L \times V_1)$.

A subgraph $G_1 = \langle V_1, L, A_1 \rangle$ of G is said to be a *strongly connected component* of G if:

1. G_1 is strongly connected, and
2. For all subsets $V_2 \subseteq V$ such that $V_2 \neq V_1$ and $V_2 \supseteq V_1$, the subgraph $G_2 = \langle V_2, L, A_2 \rangle$ is not strongly connected.

A *tree* $T = \langle V, L, A, r \rangle$ is a directed graph $\langle V, L, A \rangle$ with a distinguished *root* $r \in V$, such that for every $v \in V$ $(v \neq r)$ there is *at least* one path joining r and v.*

INTERPRETED GRAPHS

An *interpreted graph* IG consists of a directed graph $\langle V, L, A \rangle$, where

1. With each vertex $v \in V$, there is associated a domain D_v, and
2. With each arc $a = (v, l, v') \in A$, there is associated a total *test predicate* P_a which maps D_v into $\{T, F\}$, and a total *function* f_a which maps $D_v{}^*$ into $D_{v'}$, where $D_v{}^* = \{x \mid x \in D_v \wedge P_a(x) = T\}$.

Let $(v_o, x_0) \in V \times D_{v_o}$ be an *arbitrary* pair of an interpreted graph IG. A (v_0, x_0)-*execution sequence* of IG is a (finite or infinite) sequence of the form

$$(v^{(0)}, x^{(0)}) \xrightarrow{l^{(0)}} (v^{(1)}, x^{(1)}) \xrightarrow{l^{(1)}} (v^{(2)}, x^{(2)}) \xrightarrow{l^{(2)}} \cdots,$$

where

1. $v^{(j)} \in V$, $l^{(j)} \in L$ and $x^{(j)} \in D_{v^{(j)}}$ for all $j \geq 0$;
2. $(v^{(0)}, x^{(0)})$ is (v_0, x_0);
3. If $(v^{(j)}, x^{(j)}) \xrightarrow{l^{(j)}} (v^{(j+1)}, x^{(j+1)})$ is in the sequence, then there exists an arc $a = (v^{(j)}, l^{(j)}, v^{(j+1)}) \in A$ such that $P_a x^{(j)} = T$ and $f_a x^{(j)} = x^{(j+1)}$;
4. If the sequence is finite and the last pair in the sequence is $(v^{(n)}, x^{(n)})$, then for all arcs $a \in A$ leading from $v^{(n)}$: $P_a x^{(n)} = F$.

The definition of an interpreted graph IG allows the existence of a vertex $v \in V$, an $x \in D_v$, and two *distinct* arcs a, $b \in A$ leading from v—such that both $P_a x = T$ and $P_b x = T$, i.e., the predicates on all arcs leading from the vertex v are not necessarily mutually exclusive. It follows that for the fixed pair $(v_0, x_0) \in VXD_{v_0}$, there may exist many distinct (v_0, x_0)-execution sequences of IG. For this reason, the execution process of an interpreted graph, starting with the pair (v_0, x_0), is best described by a tree.

The *execution tree* $T(v_0, x_0)$ of IG is the tree $\langle V', L, A', (v_0, x_0) \rangle$, where:

1. The set of vertices V' is the set of all pairs $(v, x) \in V \times D_v$ occurring in some (v_0, x_0)-execution sequence of IG.
2. L is the set of labels of IG;
3. The set of arcs A' is the set of all triples $((v, x), l, (v', y)) \in V' \times L \times V'$, such that $(v, x) \xrightarrow{l} (v', y)$ occurs in some (v_0, x_0)-execution sequence of IG; and
4. $(v_0, x_0) \in V'$ is the root vertex of the tree.

Example

Let us consider the interpreted graph IG^* (Figure 1), where $D_{v_1} = D_{v_2} = \{\text{the integers}\}$. There are three

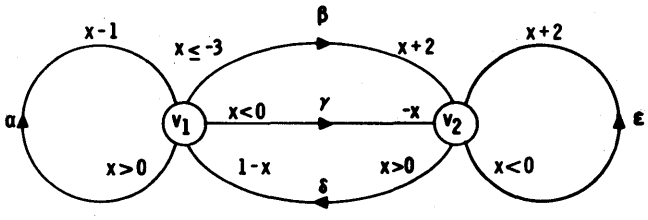


Figure 1—The interpreted graph IG^*

* Note that the standard definition of a tree has the restriction that for every $v \in V$ $(v \neq r)$ there must be *exactly* one path joining r and v.

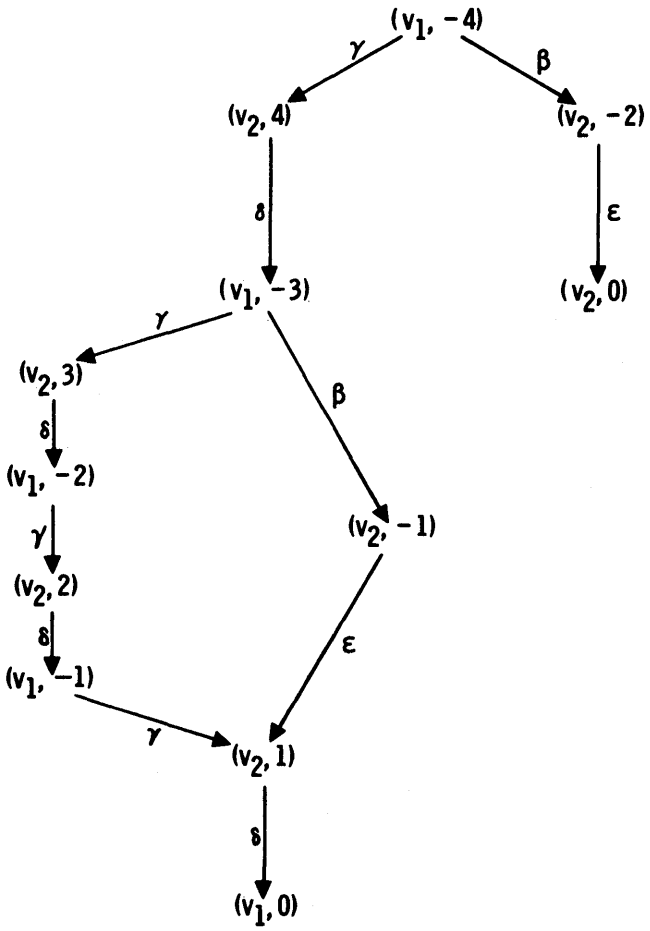


Figure 2—The execution tree $T(v_1, -4)$ of IG^*

$(v_1, -4)$-execution sequences of IG^*, namely

$$(v_1, -4) \xrightarrow{\beta} (v_2, -2) \xrightarrow{\epsilon} (v_2, 0),$$

$$(v_1, -4) \xrightarrow{\gamma} (v_2, 4) \xrightarrow{\delta} (v_1, -3) \xrightarrow{\beta} (v_2, -1) \xrightarrow{\epsilon} (v_2, 1)$$
$$\xrightarrow{\delta} (v_1, 0),$$

and

$$(v_1, -4) \xrightarrow{\gamma} (v_2, 4) \xrightarrow{\delta} (v_1, -3) \xrightarrow{\gamma} (v_2, 3) \xrightarrow{\delta} (v_1, -2)$$
$$\xrightarrow{\gamma} (v_2, 2) \xrightarrow{\delta} (v_1, -1) \xrightarrow{\gamma} (v_2, 1) \xrightarrow{\delta} (v_1, 0).$$

The execution tree $T(v_1, -4)$ of IG^* is presented in Figure 2.

TERMINATION OF INTERPRETED GRAPHS

Definition

An interpreted graph is said to *terminate* if all its execution sequences are finite, i.e., for every pair $(v_0, x_0) \in V \times D_{v_0}$ all the (v_0, x_0)-execution sequences are finite.

Notations

Let $\alpha = (a_1, a_2, \cdots, a_q)$, where $a_j = (v^{(j)}, l^{(j)}, v^{(j+1)}) \in A$ for $1 \leq j \leq q$, be any path of an interpreted graph. Then let

1. $f_\alpha(x)$ stand for $f_{a_q}(\cdots(f_{a_2}(f_{a_1}(x)))\cdots)$, and
2. $P_\alpha(x)$ stand for

$$x \in D_{v^{(1)}} \wedge P_{a_1}(x)$$
$$\wedge \bigwedge_{j=2}^{q} P_{a_j}\left(f_{a_{j-1}}(f_{a_{j-2}}(\cdots(f_{a_2}(f_{a_1}(x)))\cdots))\right)$$
$$\wedge f_\alpha(x) \in D_{v^{(q+1)}}.$$

Theorem 1

Let IG be an interpreted graph. If there exist:

1. A cut set V^* of the vertices V of IG, and
2. For every vertex $v \in V^*$, a well-ordered set $W_v = (S_v, \succ_v)$ and a total function F_v which maps D_v into S_v,

such that,

3. For every cycle α of IG:

$$v^{(1)} \xrightarrow{l^{(1)}} v^{(2)} \xrightarrow{l^{(2)}} v^{(3)} \cdots v^{(q-1)} \xrightarrow{l^{(q-1)}} v^{(q)} \xrightarrow{l^{(q)}} v^{(1)}$$

(where $v^{(1)} \in V^*$ and $v^{(k)} \neq v^{(1)}$ for all $1 < k \leq q$), and for every x such that $P_\alpha(x) = T$:

$$F_{v^{(1)}}(x) \succ_{v^{(1)}} F_{v^{(1)}}(f_\alpha(x))$$

then IG terminates.*

Proof

Proof by contradiction.

Let us assume that IG does not terminate, i.e., there exists an infinite execution sequence γ in IG,

$$\gamma: (v^{(0)}, x^{(0)}) \xrightarrow{l^{(0)}} (v^{(1)}, x^{(1)}) \xrightarrow{l^{(1)}} (v^{(2)}, x^{(2)}) \xrightarrow{l^{(2)}} \cdots.$$

Let γ' be the infinite path

$$\gamma': v^{(0)} \xrightarrow{l^{(0)}} v^{(1)} \xrightarrow{l^{(1)}} v^{(2)} \xrightarrow{l^{(2)}} \cdots.$$

Since IG, by definition, contains a finite set of vertices, and V^* is a cut set, it follows that there

* In Manna[2] it is proved, by the use of König's Infinity Lemma, that if IG consists of a finite directed graph, then this is also a necessary condition for the termination of IG.

exists a vertex $v^* \in V^*$ that occurs infinitely many times in γ'.

Let $v^{(n_1)}, v^{(n_2)}, v^{(n_3)}, \cdots (0 \leq n_j < n_{j+1}$ for $j \geq 1)$ be the infinite sequence of all occurrences of the vertex v^* in γ'. Therefore, the infinite execution sequence γ can be written as

$$\gamma\colon (v^{(0)}, x^{(0)}) \xrightarrow{l^{(0)}} \cdots (v^{(n_1)}, x^{(n_1)}) \xrightarrow{l^{(n_1)}} \cdots$$

$$(v^{(n_2)}, x^{(n_2)}) \xrightarrow{l^{(n_2)}} \cdots (v^{(n_3)}, x^{(n_3)}) \xrightarrow{l^{(n_3)}} \cdots.$$

Then, by condition (3) it follows that

$$F_v^*(x^{(n_1)}) \succ_v^* F_v^*(x^{(n_2)}) \succ_v^* F_v^*(x^{(n_3)}) \succ_v^* \cdots,$$

i.e., there is an infinite decreasing sequence in W_v^*. But this contradicts the fact that W_v^* is a well-ordered set. q.e.d.

The following corollaries follow directly from Theorem 1.

Corollary 1

Let IG be an interpreted graph. If there exist:

1. A cut set V^* of the vertices V of IG,
2. A well-ordered set $W = (S, \succ)$, and
3. For every vertex $v \in V^*$, a total function F_v that maps D_v into S,

such that

4. For every elementary path α of IG:

$$v^{(1)} \xrightarrow{l^{(1)}} v^{(2)} \xrightarrow{l^{(2)}} v^{(3)} \cdots v^{(q-1)} \xrightarrow{l^{(q-1)}} v^{(q)}$$

(where $v^{(1)}, v^{(q)} \in V^*$ and $v^{(j)} \notin V^*$ for all j, $1 < j < q$), and for every x such that $P_\alpha(x) = T$: $F_{v^{(1)}}(x) \succ F_{v^{(q)}}(f_\alpha(x))$,

then IG terminates.

Corollary 2

Let IG be an interpreted graph, which has a vertex v^* common to all its (elementary) cycles.

If there exist a well-ordered set $W = (S, \succ)$ and a total function F which maps D_{v^*} into S, such that for every elementary cycle $\alpha\colon v^* \to \cdots \to v^*$ and for every x such that $P_\alpha(x) = T\colon F(x) \succ F(f_\alpha(x))$, *then* IG terminates.

Definition

Let IG be an interpreted graph constructed from the directed graph G.

Then a *strongly connected component* IG' *of* IG consists of a strongly connected component $G' = \langle V', L, A' \rangle$ of G, where,

1. With each vertex $v \in V'$, there is associated the domain D_v of IG and
2. With each arc $a \in A'$, there is associated the test predicate P_a and the function f_a of IG.

Theorem 2

An interpreted graph IG terminates *if and only if* all its strongly connected components terminate.

Proof

($\Rightarrow$) Follows directly from the definition of termination of interpreted graphs.

($\Leftarrow$) Proof by contradiction.

Let us assume that IG does not terminate, i.e., there exists an infinite execution sequence γ in IG,

$$\gamma\colon (v^{(0)}, x^{(0)}) \xrightarrow{l^{(0)}} (v^{(1)}, x^{(1)}) \xrightarrow{l^{(1)}} (v^{(2)}, x^{(2)}) \xrightarrow{l^{(2)}} \cdots.$$

Let γ' be the infinite path

$$\gamma'\colon v^{(0)} \xrightarrow{l^{(0)}} v^{(1)} \xrightarrow{l^{(1)}} v^{(2)} \xrightarrow{l^{(2)}} \cdots.$$

Since IG, by definition, contains a finite set of vertices, it follows that there exist finitely many vertices of G that meet γ' only a finite number of times. Let $v^{(n_1)}, v^{(n_2)}, \cdots, v^{(n_q)}$ $(0 \leq n_j < n_{j+1}$ for $1 \leq j < q)$ be the list of their occurrences in γ'.

It follows that all the vertices $v^{(j)}$ $(j > n_q)$ of γ' are in some strongly connected component G' of G.

This implies that there exists a strongly connected component IG' of IG such that the infinite subsequence of γ:

$$(v^{(n_{q+1})}, x^{(n_{q+1})}) \xrightarrow{l^{(n_{q+1})}} (v^{(n_{q+2})}, x^{(n_{q+2})}) \xrightarrow{l^{(n_{q+2})}} \cdots$$

is an infinite execution sequence of IG', i.e., IG' does not terminate. Contradiction. q.e.d.

APPLICATIONS

The results of the preceding section can be used for proving termination of various classes of algorithms. In this section we shall illustrate the use of the results for proving termination of programs and recursively defined functions.

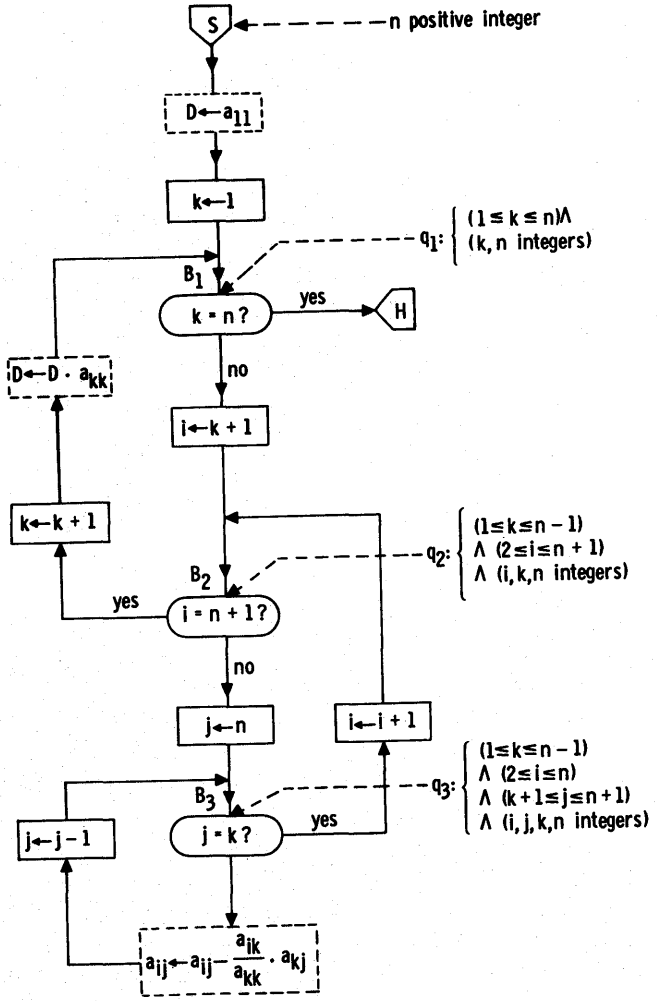


Figure 3—A program for evaluating a determinant $|a_{ij}|$ of order n, $n \geq 1$, by Gaussian elimination

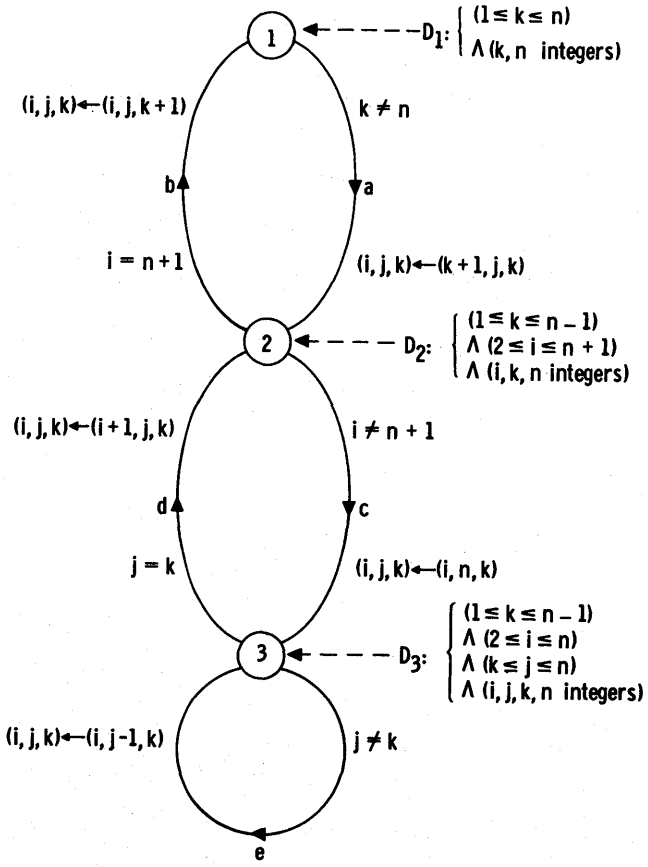


Figure 4—The interpreted graph IG_1

Example 1:

Consider the program (Figure 3)* for evaluating a determinant $|a_{ij}|$ of order n, $n \geq 1$, by Gaussian elimination.** Here n is an integer constant, $(a_{ij})_{1\leq i,j\leq n}$ a real array, D a real variable, and i, j, k integer variables. We want to show that the program terminates for every positive integer n.

Since neither D nor any a_{ij} occurs in a test box or affects the value of any variable that occurs in a test box, it is clear that by erasing the three assignments, denoted by dashed boxes in Figure 3, we do not change the termination properties of the program.

One can verify easily that the set of predicates attached to the test boxes of the flowchart is a valid interpretation with respect to the initial predicate "n positive integer"; i.e., starting with any initial positive integer n, whenever the flow of control through the flowchart reaches the test box B_i the current values of the variables satisfy the predicate q_i (see Floyd[1]).

Let us construct now, from the reduced program (Figure 3 without the dashed boxes), the appropriate interpreted graph IG_1 (Figure 4), such that each vertex v_i, $1 \leq i \leq 3$, of Figure 4 corresponds to the test box B_i of Figure 3, and its domain D_i is exactly the valid interpretation q_i. Note that we have used Theorem 2 here, since we consider only the strongly connected component of our graph.

It is clear that, if the interpreted graph IG_1 terminates, then the given program terminates for every positive integer n. But the termination of IG_1 follows from Corollary 1, where

$$V^* = \{2, 3\} \text{ is the cut set,}$$

$$W = I_3^+ \text{ is the well-ordered set,}$$

$$F_2(i, j, k) = (n - 1 - k, n + 1 - i, n + 1)$$

is the mapping of D_2 into W, and

$$F_3(i, j, k) = (n - 1 - k, n + 1 - i, j)$$

is the mapping of D_3 into W.

* Ignore for a moment the predicates q_1, q_2 and q_3 associated with the test boxes.

** We consider the division operator over the real domain as a total function. (Interpret, for example, $r/0$ as $r/10^{-10}$ for every real r.)

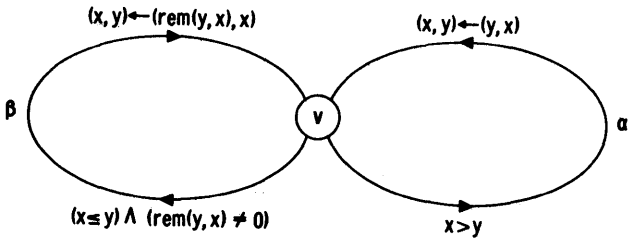


Figure 5—The interpreted graph IG_2

Example 2:

Consider the function $gcd(x, y)$ (McCarthy[3]). $gcd(x, y)$ computes the greatest common divisor of x and y (where x and y are positive integers), and is defined recursively using the Euclidean Algorithm by

$gcd(x, y) = \textit{if } x > y \textit{ then } gcd(y, x)$

$\textit{else if } rem(y, x) = 0 \textit{ then } x$

$\textit{else } gcd(rem(y, x), x),$

where $rem(u, v)$ is the remainder of u/v.

We want to show that for every pair (x, y) of positive integers, the recursive process for computing $gcd(x, y)$ terminates.

Consider the interpreted graph IG_2 (Figure 5) where D = {positive integers} × {positive integers}. By considering the vertex v as representing the start of the computation of *gcd*, it follows that the recursive process for computing $gcd(x, y)$ terminates for every pair of positive integers (x, y), if and only if the interpreted graph IG_2 terminates.

Since IG_2 consists only of one vertex, we may use

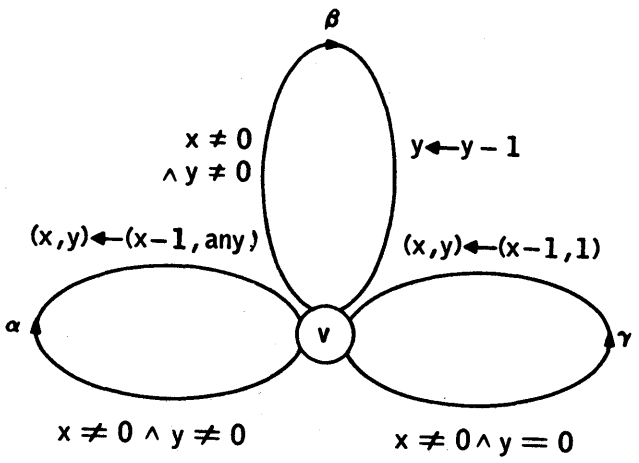


Figure 6—The interpreted graph IG_3

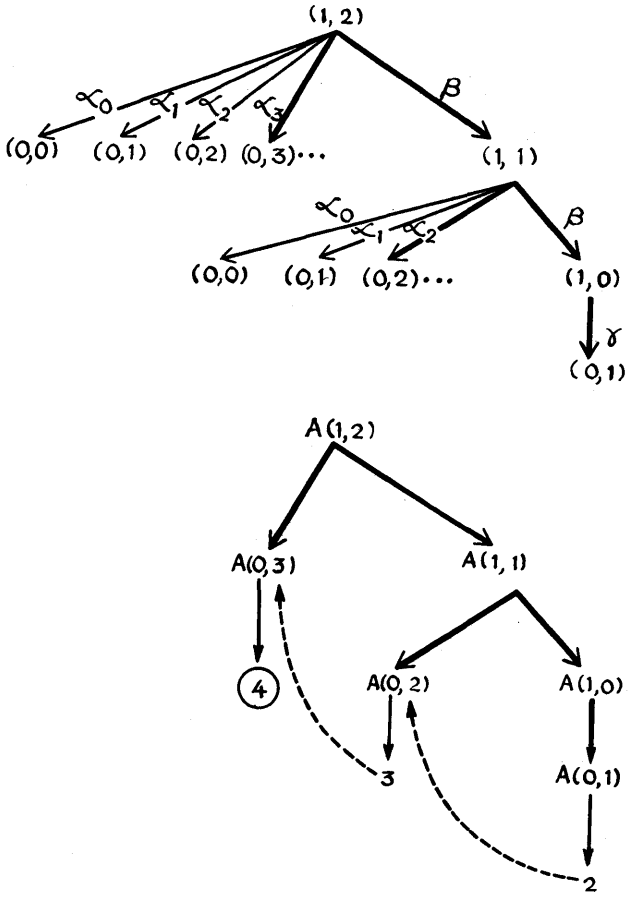


Figure 7(a)—The execution tree $T(v, (1, 2))$
(b)—The real execution tree of $A(1, 2)$

Corollary 2 to show its termination. So, let $W = I_1^+$ be the well-ordered set, and $F(x, y) = rem(y, x)$ the mapping of D into W. Since*

1. $P_\alpha(x, y) = T \Rightarrow (x, y) \in D \wedge (x > y)$

$\Rightarrow (rem(y, x) = y)$

$\wedge (y > rem(x, y) \geq 0)$

$\Rightarrow rem(y, x) > rem(x, y)$

$\Rightarrow F(x, y) > F(y, x)$, and

2. $P_\beta(x, y) = T \Rightarrow (rem(y, x), x) \in D$

$\Rightarrow rem(y, x) > rem(x, rem(y, x))$

$\Rightarrow F(x, y) > F(rem(y, x), x),$

it follows by Corollary 2 that the interpreted graph IG_2 terminates, which implies the desired result.

* Note that for every non-negative integer x, and for every positive z: $z > rem(x,z) \geq 0$.

Example 3:

Ackermann's function $A(x, y)$, where x and y are non-negative integers, is defined recursively by:

$$A(0, y) = y + 1$$

$$A(x + 1, 0) = A(x, 1)$$

$$A(x + 1, y + 1) = A(x, A(x + 1, y)).$$

We want to show that for every pair (x, y) of non-negative integers, the recursive process for computing $A(x, y)$ terminates.

Let us consider the interpreted graph IG_3 (Figure 6), where D = {non-negative integers} $\times$ {non-negative integers}. The arc α represents infinitely many arcs $\alpha_0, \alpha_1, \alpha_2, \cdots$ leading from vertex v to vertex v and with each arc α_i, $i \geq 0$, there is associated the test predicate $x \neq 0 \wedge y \neq 0$ and the function $(x, y) \leftarrow (x - 1, i)$.

In other words, 'any' represents all the non-negative integers and therefore includes all possible values of $A(x + 1, y)$. It follows that, for every pair (x, y) of non-negative integers, the execution tree $T(v, (x, y))$ of IG_3 (i.e., execution starts from v with (x, y)) contains the real execution tree of $A(x, y)$ as a "subtree". This is illustrated in Figure 7 for $A(1, 2)$.

This implies that if the interpreted graph IG_3 termites, then the recursive process for computing $A(x, y)$ terminates (for every pair (x, y) of non-negative integers). But the termination of the interpreted graph follows clearly from Corollary 2, where

$W = I_2^+$ is the well-ordered set, and

$F(x, y) = (x, y)$ is the mapping of D into W.

ACKNOWLEDGMENTS

I am indebted to Robert W. Floyd for his help and encouragement throughout this research. I am also grateful to Stephen Ness for his detailed reading of the manuscript.

REFERENCES

1 R W FLOYD
Assigning meaning to programs
Proceedings of Symposia in Applied Mathematics American Mathematical Society Vol 19 pp 19–32 1967

2 Z MANNA
Termination of algorithms
PhD Thesis Computer Science Department Carnegie-Mellon University April 1968

3 J McCARTHY
Recursive functions of symbolic expressions and their computation by machine part I
Communications of the ACM Vol 3 No 4 pp 184–195 April 1960

A planarity algorithm based on the Kuratowski theorem*

by PENG-SIU MEI

Honeywell, Inc.
Wellesley Hills, Massachusetts

and

NORMAN E. GIBBS

College of William and Mary
Williamsburg, Virginia

INTRODUCTION

In the layout of integrated circuits and printed circuits, one often wants to know if a particular electrical network is planar, i.e., can be imbedded in the plane without having any line crossing another line. Our algorithm, when given a finite graph G can decide if G is planar. The algorithm was implemented in Fortran together with the Cycle Generation Algorithm for Finite Undirected Linear Graphs of Gibbs[3] and used extensively to test the planarity of a large number of graphs. The distinguishing characteristic of this algorithm is its conceptual simplicity and its ease of implementation on a computer. The computer program took but a few days to write and debug.

In contrast to some of the recent work on the same subject,[1,2,6,7] this planarity algorithm is a direct application of the Kuratowski Theorem. It is based on the observation that a Kuratowski graph can be spanned by the union of two of its circuits. This algorithm can be used in conjunction with existing algorithms which generate all the circuits of a given graph G. After obtaining all the circuits of G, our algorithm examines the subgraphs spanned by pairs of circuits to see if these subgraphs contain a Kuratowski graph.

The paper begins with the necessary notation and definitions. This is followed by the presentation of the algorithm and a few brief comments.

Definition 1: Let V be a finite non-empty set and $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V \text{ and } v_1 \neq v_2\}$, then $G = \langle V, E\rangle$ is a *finite undirected graph without loops or multiple edges*, or more simply, a *graph*.

Definition 2: A *subgraph* $G' = \langle V', E'\rangle$ of a graph $G = \langle V, E\rangle$ is a graph where $V' \subseteq V$ and $E' \subseteq E$.

Definition 3: Let $G = \langle V, E\rangle$ be a graph and X a non-empty subset of E, then $S_G(X) = \langle V', X\rangle$, the *subgraph of G spanned by X*, is the subgraph of G where $V' = \{v \mid v \in V \text{ and for some } x \in X, v \in x\}$.

Definition 4: A non-empty subset C of edges of a graph G is a *circuit* (or *cycle*) of G if $S_G(C) = \langle V', C\rangle$ is such that for each $v \in V'$, there are exactly two elements of C which contain v, and C does not properly contain any other circuit of G. C is said to be of *length k* if it has k elements.

Definition 5: The class of all subgraphs of G which are spanned by the union of two distinct circuits of G will be denoted by $TC(G)$, i.e.,

$$TC(G) = \{S_G(C_1 \cup C_2) \mid C_1 \text{ and } C_2 \text{ are distinct circuits of } G\}$$

Definition 6: Let $G = \langle V, E\rangle$ be a graph, we define an *open simple path of G* inductively. $\langle \phi, \{e\}\rangle$, where $e \in E$, is an open simple path. If $\langle V', E'\rangle$ is an open simple path, then so is $\langle V' \cup \{v\}, E' \cup \{e\}\rangle$ where

1. $e \in E - E'$
2. $v \in V - V'$
3. there is some $e' \in E'$ such that $v \in e \cap e'$
4. for all $v' \in V', v' \notin e$.

Figure 1 shows two examples of open simple paths.

Definition 7: Let $\langle V, E\rangle$ be an open simple path of G and $\langle V_1, E\rangle = S_G(E)$. Then $V_1 - V$ has exactly

* This work was supported in part by the National Science Foundation Grant GJ-120 and a Purdue David Ross Research Grant.

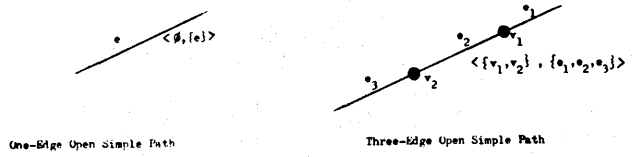


Figure 1

two elements, u and v, and we say *u and v are connected by the open simple path* $\langle V, E\rangle$.

Definition 8: Two open simple paths $\langle V', E'\rangle$ and $\langle V'', E''\rangle$ are *disjoint* if and only if $V' \cap V'' = E' \cap E'' = \phi$.

Definition 9: A K_5* *graph* is a graph which can be constructed by taking a set V of five vertices and connecting every pair of distinct elements of V by an open simple path such that these open simple paths are pairwise disjoint.

Definition 10: A $K^*_{3,3}$ *graph* is a graph which can be constructed by taking two disjoint sets, V_1 and V_2 of three vertices each and connecting every member of V_1 to every member of V_2 by an open simple path such that these open simple paths are pairwise disjoint.

Figure 2 shows examples of the simplest K_5* and $K^*_{3,3}$ graphs.

Note that every K_5* (every $K^*_{3,3}$) graph may be obtained from that of Figure 2 by replacing the set of edges by a set of pairwise disjoint open simple paths.

Theorem (Kuratowski[5]): A graph is planar if and only if it does not have a subgraph which is a K_5* or a $K^*_{3,3}$ graph.

Observation (J. R. Buchi): A graph G is planar if and only if $TC(G)$ does not contain any K_5* or $K^*_{3,3}$ graphs.

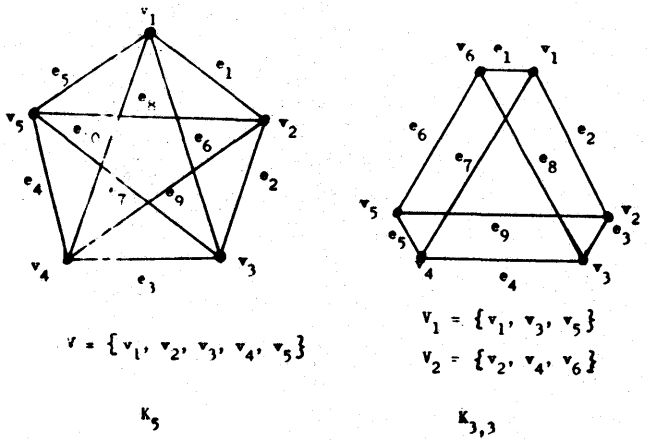


Figure 2

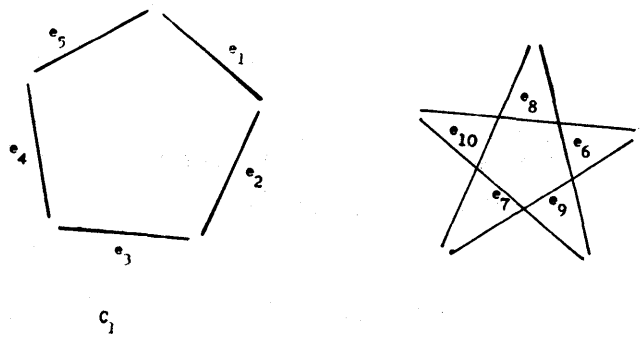


Figure 3

In fact this follows from the Kuratowski Theorem because each K_5* and each $K^*_{3,3}$ can be spanned by two circuits. The union of the two circuits in Figure 3 span the K_5* graph of Figure 2.

The union of the two circuits of Figure 4 span the $K^*_{3,3}$ graph of Figure 2.

We are now ready to state our algorithm which may be programmed in conjunction with a circuit generation algorithm, for example, the algorithms of Gotlieb and Corneil,[4] and of Gibbs,[3] to determine whether or not a graph G is planar from its vertex-adjacency matrix (vertices vs. vertices). Given two circuits C_1 and C_2, let $S_G(C_1) = \langle V_1, C_1\rangle$ and $S_G(C_2) = \langle V_2, C_2\rangle$. In brief, steps 2 and 3 of the algorithm check to see if $S_G(C_1 \cup C_2)$ is a K^*_5 graph. If $S_G(C_1 \cup C_2)$ is not a K_5* graph, $V_1 \cap V_2$ has more than five elements, and $C_1 \cap C_2$ has more than two elements, then steps 5 through 8 of the algorithm essentially eliminate all the vertices of degree 2 of $S_G(C_1 \cup C_2)$ and then check to see if the resultant graph is $K_{3,3}$—the simplest of the $K^*_{3,3}$ graphs.

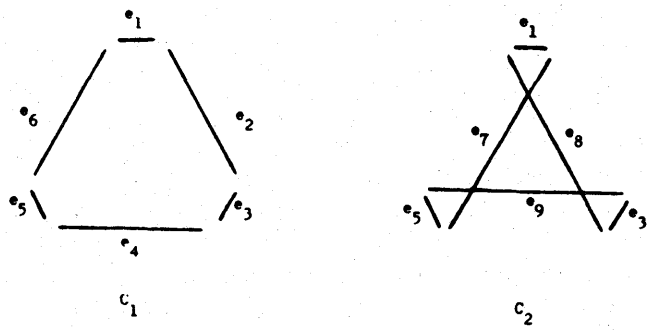


Figure 4

ALGORITHM

1. Given a graph G, generate all the circuits of length five or greater.

2. Given two circuits C_1 and C_2, let $S_G(C_1) = \langle V_1, C_1 \rangle$ and $S_G(C_2) = \langle V_2, C_2 \rangle$. If $V_1 \cap V_2$ has exactly five elements and $C_1 \cap C_2 = \phi$, go to the next step, otherwise, go to step 4.

3. Trace $S_G(C_1)$ in one direction and let $(v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}, v_{i_5})$ be the elements of $V_1 \cap V_2$ ordered in this cyclic order. Check to see if these elements can be placed in a cyclic order $(v_{i_1}, v_{i_3}, v_{i_5}, v_{i_2}, v_{i_4})$ when $S_G(C_2)$ is traced. If the answer is "yes," $S_G(C_1 \cup C_2)$ is a K_5* graph and G is non-planar. If the answer is "no," go to step 9.

4. If $V_1 \cap V_2$ has more than five elements and $C_1 \cap C_2$ has more than two elements, go to the next step, otherwise, go to step 9.

5. Form the vertex-adjacency matrix $M = (m_{ij})$ of $S_G(C_1 \cup C_2)$ as follows:

$$m_{ij} = \begin{cases} 0 & \text{if } \{v_i, v_j\} \notin C_1 \cup C_2 \\ 1 & \text{if } \{v_i, v_j\} \in C_1 - C_2 \\ 2 & \text{if } \{v_i, v_j\} \in C_2 - C_1 \\ 3 & \text{if } \{v_i, v_j\} \in C_1 \cap C_2 \end{cases}$$

6. Go through the matrix row by row once, doing the following:

If row k has exactly two non-zero entries (note that these must be equal), say m_{ki} and m_{kj} are not zero, then add m_{ki} to m_{ij}, and m_{ji} and set m_{ki}, m_{ik}, m_{kj}, and m_{jk} to zero. Otherwise, go to the next row.

7. After the last row, if there remain exactly six rows with non-zero entries and each of these rows has exactly three non-zero entries, go to step 8, otherwise, go to step 9.

8. The resultant matrix is the vertex-adjacency matrix of a cubic graph $G' = \langle V', E' \rangle$ with six vertices. Let C_1' be the circuit of G' consisting of the six edges labeled by a "1" or a "3" and let C_2' be the circuit of G' consisting of the six edges labeled by a "2" or a "3." Note that $S_{G'}(C_1' \cup C_2') = G'$. Let $(v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}, v_{i_5}, v_{i_6})$ be the elements of V' in the cyclic order obtained by tracing $S_{G'}(C_1')$ in one direction with the edge $\{v_{i_1}, v_{i_2}\} \in C_1' \cap C_2'$. Now start with v_{i_1} and go to v_{i_2} and continue tracing $S_{G'}(C_2')$. If the resultant cyclic order of V' is $(v_{i_1}, v_{i_2}, v_{i_5}, v_{i_6}, v_{i_3}, v_{i_4})$, then $S_G(C_1 \cup C_2)$ contains a $K^*_{3,3}$ graph, otherwise, go to step 9.

9. The graph G is planar if there are no more pairs of circuits to be considered. Otherwise, select another pair of circuits C_1 and C_2 of G, and go to step 2.

CONCLUSION

It may take the algorithm a relatively long time to find out that a large planar graph is indeed planar, but the relative ease with which the algorithm can be programmed should render it suitable for testing a small number of graphs or graphs that do not have a large number of circuits. Although a relatively large computer was used in our implementation, the algorithm is simple enough to be implemented on a computer of almost any size. Step 1 (the generation of circuits) of the algorithm can be executed first and the generated circuits can be stored on some form of auxiliary storage. The check for planarity can then be executed separately.

REFERENCES

1 AUSLANDER S V PARTER
On imbedding graphs in the sphere
J Math and Mech Vol 10 pp 517–523 1961

2 G J FISHER O WING
An algorithm for testing planar graphs from the incidence matrix
Proc 7th Midwest Symp on Circuit Theory Ann Arbor Michigan May 1964

3 N E GIBBS
A cycle generation algorithm for finite undirected linear graphs
Journal of the ACM Vol 16 No 4 pp 564–568 October 1969

4 C C GOTLIEB D G CORNEIL
Algorithms for finding a fundamental set of cycles for an undirected linear graph
Communications of the ACM Vol 10 No 12 pp 780–783 December 1967

5 G KURATOWSKI
Sur le problem des Courbes Gauches en Topologie
Fund Math Vol 15 pp 271–283 1930

6 A LEMPEL S EVEN I CEDERBAUM
An algorithm for planarity testing of graphs
Theory of Graphs International Symposium at Rome Gordon and Breach New York 1967

7 P M LIN
On methods of detecting planar graphs
Proc 8th Midwest Symposium on Circuit Theory Colorado State University June 1966

8 S MACLANE
A combinatorial condition for planar graphs
Fund Math Vol 28 pp 22–32 1937

9 H WHITNEY
Non-separable and planar graphs
Trans Am Math Society Vol 34 pp 339–362 1932

Combinational arithmetic systems for the approximation of functions*

by CHIN TUNG

IBM Research Laboratory
San Jose, California

and

ALGIRDAS AVIZIENIS

University of California
Los Angeles, California

INTRODUCTION

The concepts of arithmetic building blocks (ABB) and combinational arithmetic (CA) nets as well as their applications have been previously reported in References 3, 4, and 5. The unique ABB, resulting from the efforts of minimizing the set of building blocks in Reference 3, is designed at the arithmetic level, employing the redundant signed-digit number system,[2] and is to be implemented as one package by LSI techniques. The ABB performs arithmetic operations on individual digits of radix $r > 2$ and its main transfer functions are: the sum (symbol $+$) and product (symbol $*$) of two digits, the multiple sum of m digits ($m \leq r + 1$), (symbol $\neq$), and the reconversion to a non-redundant form (symbol RS).

A single ABB may serve as the arithmetic processor of a serially organized computer. Many ABB's can be interconnected to form parallel arrays called *combinational arithmetic* (CA) *nets* which compute sums, products, quotients, or evaluate more complex functions: trigonometric, exponential, logarithmic, gamma, etc. Because of the use of signed-digit numbers, the parallel addition and multiplication speed is independent of the length of operands. A design procedure has been developed for CA nets[5]—a given algorithm is initially represented by a directed graph (algorithm graph, or A-graph), which is then converted to an interconnected diagram of ABB's (hardware graph, or H-graph). The delay through one ABB is defined to be one time unit, Δt.

* The work was sponsored by AEC-AT(11-1) Gen. 10, Project 14.

A simple example—evaluation of polynomials—is used here to illustrate the concept of CA nets.

The method suggested by Estrin,[7] computing

$$P_n(x) = a_0 + a_1x + x^2(a_2 + a_3x) + x^4(a_4 + a_5x + x^2(a_6 + a_7x)) + \cdots$$

permits the fastest evaluation when CA nets are used. This is shown in Figure 1 with $n = 3$; the extension to higher values of n is evident. In general, the delay through such a net is $\lfloor \log_2 n \rfloor + 1$ multiplication–addition times.

This paper summarizes our study of applying a particular version of CA nets, i.e., pipelined CA nets, to approximating functions. Involved are not only the topological layout of pipelined CA nets for approximating functions but also the computational complexity.

Throughout this paper, we will use minimally redundant radix 16 signed-digit number representation whose allowed digit values are $\{-9, -8, \cdots, -1, 0, 1, \cdots, 9\}$.

APPROXIMATION OF FUNCTIONS

The basic capability of a typical digital computer is limited to simple algebraic manipulations. As a result of this inherent limitation approximation is inevitably involved in the practical computational procedure if the numerical approach is to apply to the evaluation of functions at all. The discrepancy between approximated and the approximating values is required to be

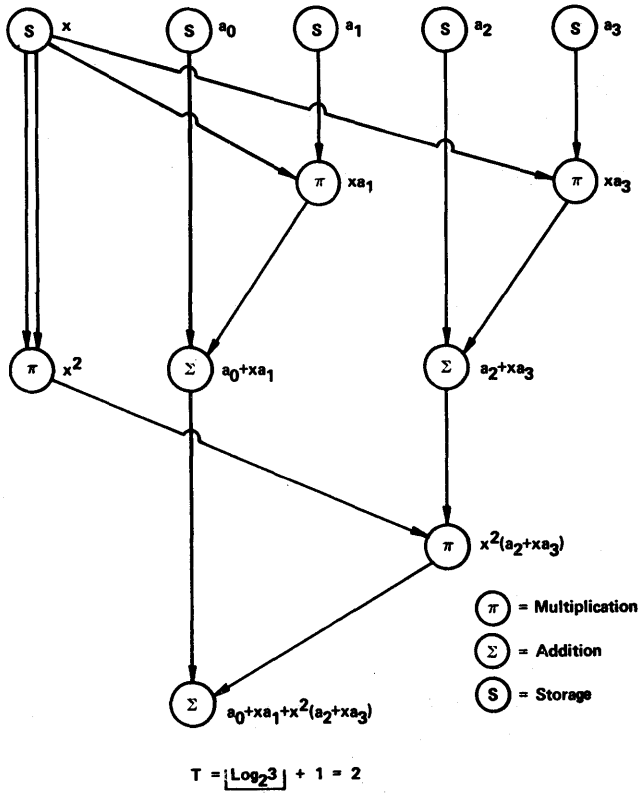


Figure 1—Evaluation of 3rd degree polynomial with Estrin's method

adjusted to a certain tolerable degree as individual cases demand. There are two general approaches in the theory of approximation—polynomial approximation and rational approximation.[8]

The representation of functions by polynomials is an old art. The Taylor series has been one of the cornerstones of analytical research. If a series has no other purpose than numerical evaluation of the function, the degree of convergence has to be investigated. The Taylor expansion may converge in the entire plane or within a given circle only, and it may diverge even at every point. With the development of the theory of orthogonal expansions, the realization came that occasionally power expansions whose coefficients are not determined according to the scheme of Taylor expansion can operate more effectively than Taylor series itself. Such expansions are not based on the process of successive differentiation but on integration. A large class of functions which are not sufficiently analytic to allow a Taylor expansion can be represented by such orthogonal expansions. These expansions belong to a given definite real realm of the independent variable x, and it is aimed to approximate a function in such a way that the error shall be of the same order of magnitude all over the range. Rapidly convergent power expansions are of practical importance. Mere convergence of an expansion, valuable as it is from the purely analytical standpoint, is of little practical use if the number of terms demanded for a reasonable accuracy is very large.[9]

In light of the above consideration *Chebyshev polynomials*, defined by

$$T_n(x) = \text{Cos}\ (n\ \text{Cos}^{-1}) \quad \text{for} \quad -1 \leq x \leq +1, \quad (1)$$

emerge as a promising potential candidate for approximating functions.

With the speed of division rapidly increased in conventional computers, the superiority of rational approximations seems to be generally recognized.[9] Roughly speaking, one may say that the "curve-fitting ability" of *rational function* $R(x)$

$$R(x) = \frac{a_0 + a_1x + a_2x^2 + \cdots + a_mx^m}{b_0 + b_1x + b_2x^2 + \cdots + b_nx^n}$$

is approximately equal to that of a polynomial of degree $n + m$. In competing with the polynomial of degree $n + m$, $R(x)$ has an unsuspected advantage in that the computation of $R(x)$ for a given x does not require $n + m$ additions, $n + m - 1$ multiplications, and one division as might be surmised at first. By transforming $R(x)$ into a *continued fraction*

$$R(x) = P_1(x) + \cfrac{C_2}{P_2(x) + \cfrac{C_3}{P_3(x) + \cdots}} \quad \cdots \quad + \frac{C_k}{P_k(x)} \quad (2)$$

we achieve the significant reduction in the number of multiplications and divisions for evaluating any $R(x)$ to n or m, whichever is larger. The continued fraction form of a rational function not only lends itself to a faster execution but also, sometimes, refrains from a disadvantage suffered by the rational functions—the coefficients depend on the degrees of the numerator and denominator.

A practical application of CA nets to approximating a given function should involve three basic criteria: *speed*, *accuracy*, and *cost*. The overall speed on a machine is governed by two factors: the speed of signals physically going through circuit components and the speed of the computational algorithm in terms

of logical steps. We are primarily concerned with the latter. The inherent unique property of a totally combinational arithmetic net clearly allows as many parallel computations to be done simultaneously as they are mathematically permissible. Therefore, the delay from the presence of given data to the interpretation of the result could be minimized in a CA net. As the evaluation of a given function through the numerical technique inevitably introduces approximations, the problem of *accuracy* is twofold. First, how accurate is the approximating formula? Second, how can the error, thus incurred, be estimated, adjusted, and controlled? *Cost* is given a restricted meaning here. It is a measure of the number of building blocks needed in the implementation.

If $f(x)$ is continuous and of bounded variation in $[-1, 1]$ then $f(x)$ can be expressed in terms of the Chebyshev series.[6]

$$f(x) = \tfrac{1}{2}a_0 + a_1T_1(x) + \cdots = \sum_{i=0}^{\infty}{}' a_iT_i(x) \tag{3}$$

This series can be truncated after any term, say nth, to give an approximation to $f(x)$. The truncation error is then

$$\sum_{i=n+1}^{\infty} | a_iT_i(x) | \leq \sum_{i=n+1}^{\infty} | a_i | \tag{4}$$

because the magnitude of $T_i(x)$ is bounded by unity. It has been shown that Chebyshev expansions are the most strongly convergent of a wide class of expansions in orthogonal polynomials.[9] Therefore, the truncation error of the Chebyshev approximation is ascertainable at a glance. Further, the partial sum of the Chebyshev series

$$\sum_{i=0}^{n}{}' a_iT_i(x) \tag{5}$$

is a good approximation to the best polynomial of degree n in $[-1, 1]$. Arguments supporting this assertion can be found in Reference 6.

Even though explicit polynomials can be evaluated on a maximally parallel CA nets, as shown in the previous section, they have some drawbacks. First, the power form given by

$$f(x) = c_{0,n} + c_{1,n}x + \cdots + c_{n,n}x^n \tag{6}$$

has coefficients which are functions of n, so that a change in order of approximation requires a new set of coefficients. The second drawback stems from the ill-determination of the coefficients $c_{i,n}$ when n is large, which frequently occurs when a function is represented to high accuracy over a long range.

Recent developments have demonstrated that rational approximations can give higher accuracy than Chebyshev approximations of the same computational complexity.[10] However, approximation of the form

$$R(x) = P(x)/Q(x) = f(x) \tag{7}$$

share one of the disadvantages of explicit polynomials; the coefficients of $P(x)$ and $Q(x)$ depend on the degrees of $P(x)$ and $Q(x)$. Continued fractions derived from the form (7) can overcome this drawback and have shown a promising prospect in numerical computation on conventional computers. Nevertheless, continued fractions are still impaired by some shortcomings, at least, as far as the application of CA nets is concerned. The most serious problem in this respect is that the evaluation of a continued fraction involves a series of divisions; division is rather complicated in a CA net.[11,12] The other shortcoming of less importance is that integration and differentiation cannot be done on a continued fraction as easily as on a Chebyshev series.

The fact that continued fractions involve many divisions has forced us to choose polynomial approximations, which have no division at all, rather than rational approximations in the design of the combinational arithmetic system for the approximation of functions. This choice is more or less unique to our system and may not be justified in many other cases. Further improvement of this system may alter this basic decision.

PIPELINED COMBINATIONAL ARITHMETIC NETS FOR EVALUATING CHEBYSHEV SERIES

In between the two extremes, totally serial and totally concurrent (e.g., Figure 1), a pipelined CA (PCA) net serves as a compromised alternative. A PCA net, in general, consists of both sequential and combinational circuits. Different composition of these two kinds of circuits gives to the resultant PCA net a wide spectrum of performance versus cost. A designer is thus endowed with more freedom at his disposal to choose a particular composition to meet his requirements. The study we made shows that the PCA net is particularly attractive for evaluating Chebyshev series. The general concept of pipelining techniques has been successfully applied to modern information processing systems in order to obtain a much improved performance at the cost of very moderate increase of hardware.[1]

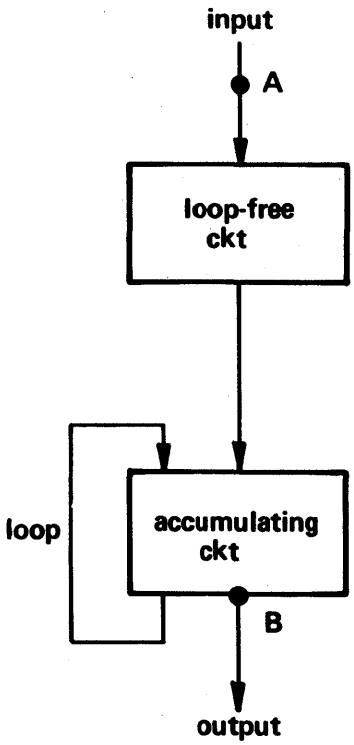


Figure 2—An abstract model of pipelining

An abstract notion of pipelining

The concept of pipelining technique relevant to the evaluation of Chebyshev series and polynomials can be abstracted with the following simplified model.

Consider

$$f = \sum_{i=0}^{n} f_i \tag{8}$$

Assume not only that the computations of f_i's are independent of one another but also that the computation pattern of each f_i is essentially the same, or can be made the same by introducing dummy operations if necessary. The block identified by "loop-free ckt" in Figure 2 assumes the responsibility of computing f_i's; the raw data of f_i's are fed into it one after the other. The computed results of f_i's are then accumulated in the "accumulating ckt."

Let $t_{0,i}$ be the instant at which the raw data for computing f_i are fed into the pipelined circuit (at point A), $t_{f,i}$ the instant at which f_i is computed and accumulated as the partial result (at point B), ΔT the amount of time needed to compute f_i, i.e., $\Delta T = t_{f,i} - t_{0,i}$, T the total amount of time required to evaluate f.

Clearly, assume

$$t_{0,0} = 0$$

then

$$t_{f,i} = t_{0,i} + \Delta T$$

$$T = t_{f,n} = t_{0,n} + \Delta T \tag{9}$$

In order to decrease T one must decrease $t_{0,n}$ or ΔT, or both. To decrease ΔT is to shorten the longest information flow path; to decrease $t_{0,n}$ is to minimize the time spacings between consecutive t_0's. The minimum possible value of $t_{0,n}$ is $n \cdot \Delta t$.

It is interesting to investigate the effect on T by variations of ΔT and the time spacing between consecutive t_0's. Suppose ΔT is increased by Δt, due to the addition of more circuit in the longest information flow path, i.e., $\Delta T' = \Delta T + \Delta t$, then the corresponding total computation time T' becomes

$$T' = t_{0,n} + \Delta T' = t_{0,n} + \Delta T + \Delta t = T + \Delta t \tag{10}$$

In most cases, T is much greater than Δt, hence T' is approximately the same as T.

On the other hand, suppose the time spacing between consecutive t_0's is increased by, Δt, i.e.,

$$\begin{aligned} t_{0,0}' &= t_{0,0} \\ t_{0,1}' &= t_{0,1} + \Delta t \\ &\vdots \\ t_{0,i}' &= t_{0,i} + i \cdot \Delta t \\ &\vdots \\ t_{0,n}' &= t_{0,n} + n \cdot \Delta t \end{aligned} \tag{11}$$

then

$$T' = t_{0,n}' + \Delta T = t_{0,n} + n \cdot \Delta t + T = T + n \cdot \Delta t \tag{12}$$

A comparison of Equations (10) and (12) clearly shows that the effect of a variation of the time spacing between consecutive t_0's is n times greater than that of a variation of ΔT with the same magnitude. Therefore, it is more desirable to decrease the time intervals between consecutive t_0's than to shorten the longest information flow path.

Ideally, one would like to see the input data for computing the f_i's are fed into the pipelined circuit one after the other with the least possible delay. In this case, with

$$t_{0,0} = 0$$

we have

$$t_{0,i} = i \cdot \Delta t,$$

$$t_{0,n} = n \cdot \Delta t,$$

and

$$T = n \cdot \Delta t + \Delta T \tag{13}$$

In Equation (13), with Δt fixed, the interactions of T, n, and ΔT, can be briefly summarized in the following. As the circuit complexity increases, most likely ΔT will be lengthened and the computational power of the circuit will be enhanced. If Equation (8) can be reorganized

$$f = \sum_{i=0}^{n'} f_i' \tag{14}$$

with the complexity of f_i' greater than that of f_i, then we expect $n' < n$. The effect on T of the increase of ΔT and decrease of n cannot be specified without detailed information. It remains to be investigated.

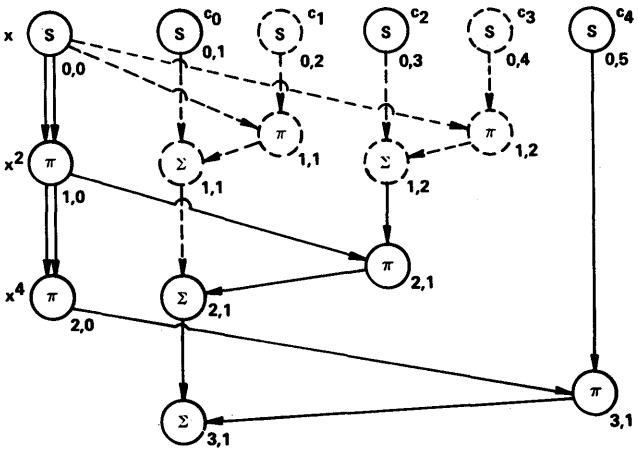


Figure 4—CA net for evaluating 4th degree polynomial

Layout of PCA nets

With the knowledge of the above section, we can now begin the layout of PCA nets for evaluating Chebyshev series. The functional block diagram of a PCA–W net is shown in Figure 3. It consists of three subnets, namely, CA–W subnet, self-multiplication CA (SMCA) subnet, and pipelined sequential CA (PSCA) subnets. A CA net with the capacity of computing polynomials $P_n(x)$, $n \leq w$, asynchronously without the necessity of segmenting $P_n(x)$ is said to have a width w and is denoted by CA–W net. The meaning of SMCA and PSCA will be clear in the later text.

The Chebyshev series used to approximate a given function $f(x)$ assumes the following forms:

$$f(x) = \sum_{i=0}^{n}{}' a_i T_i(x) \qquad -1 \leq x \leq 1$$

$$T_i(x) = \sum_{j=0}^{i/2} c_{2j} x^{2j} \qquad \text{for } i = 0, 2, 4, \cdots,$$

$$T_i(x) = \sum_{j=0}^{(i-1)/2} c_{2j+1} x^{2j+1} \qquad \text{for } i = 1, 3, 5, \cdots, \tag{16}$$

SMCA assumes the responsibility of computing the powers of x. Using Estrin's method, CA–W specializes in evaluating $T_i(x)$ when $i \leq w$. If $i > w$, then $T_i(x)$ must be broken into several segments. Each segment can be evaluated on the CA–W net with Estrin's methods. The input data for computing these segments are fed into the CA–W one immediately after the other. The outputs from both CA–W and SMCA arrive in the PSCA to form $T_i(x)$. At PSCA, the coefficient a_i is multiplied with $T_i(x)$, and $a_i T_i(x)$ must then be accumulated.

One of the inherent properties of Chebyshev polynomials, as can be seen from Equation (16), is that $T_i(x)$ contains only even terms when i is even and only odd terms when i is odd. Due to this inhomogeneity, the CA–W subnet can be simplified by removing half of the storage vertices as well as all the π–Σ pairs of vertices for evaluating the sub-expressions, $c_j + c_{j+1}x$. For instance, a full CA net for evaluating a normal 4th degree polynomial

$$P_4(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4 \tag{17}$$

is shown in Figure 4.

Since

$$T_4(x) = c_0 + c_2 x^2 + c_4 x^4 \tag{18}$$

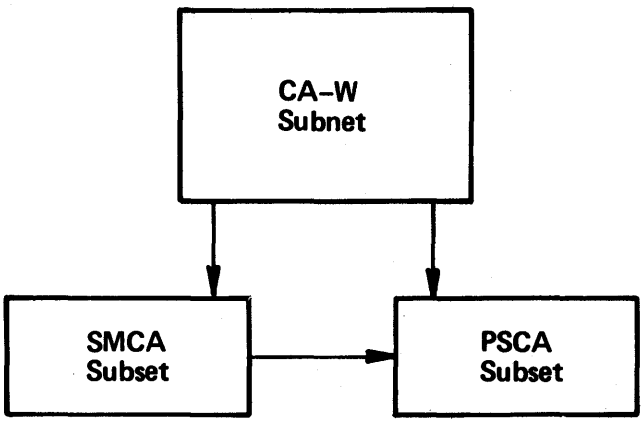


Figure 3—Functional block diagram of the PCA-W net

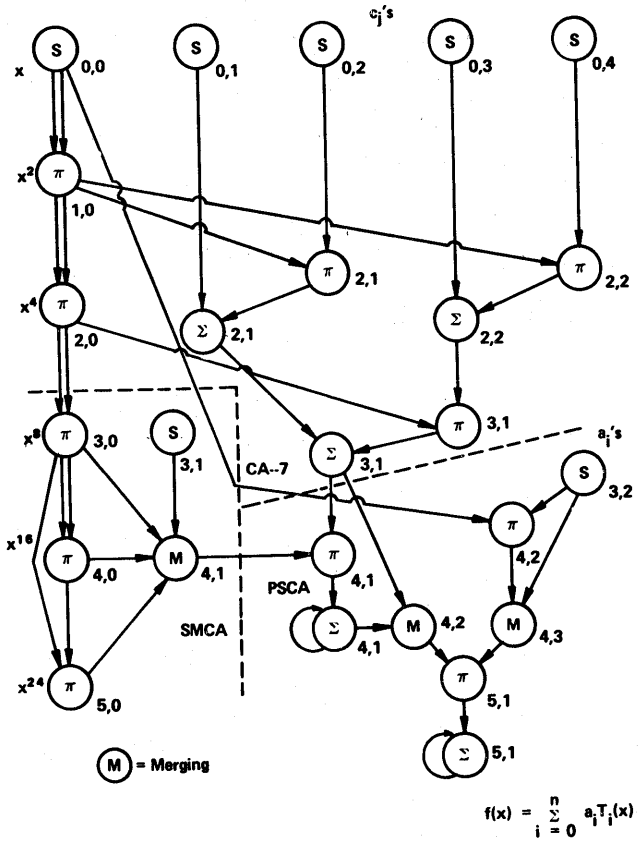


Figure 5—PCA-7 net for evaluating Chebyshev series—A-level

with $c_1 = c_3 = 0$, all the dotted vertices and arcs in Figure 4 can be removed but with $S_{0,1}$ and $S_{0,3}$ connected to $\Sigma_{2,1}$ and $\pi_{2,1}$ respectively. Roughly speaking, the cost of a CA net for evaluating a Chebyshev polynomial with degree i is about half of that of a normal polynomial with the same degree. The speed, however, remains the same since it is the formation of higher degrees of the independent variable x which determines the speed in a CA net. Odd Chebyshev polynomials can be treated as if they were even ones after x is factored out. Double subscripts are appended to coefficients, c's, with first one indicating the association of c's with $T_i(x)$ and the second one as a running index within $T_i(x)$. From own on, even i will be used in the following discussion but the arguments apply to both even i and odd i cases unless otherwise stated.

Figure 5 shows a PCA–7 net at the A-level. The CA–7 subnet can handle Chebyshev polynomials with degrees up to seven. If the degree is higher than seven, then the Chebyshev polynomial is broken into segments. An example showing how this is done is given below. Assume

$$f(x) = \sum_{i=0}^{15} a_i T_i(x) = \sum_{i=0}^{15} a_i \left(\sum_{j=0}^{i/2} c_{i,2j} x^{2j} \right) \quad \text{for} \quad i = 0, 2, \cdots$$

$$+ \sum_{i=0}^{15} a_i x \left(\sum_{j=0}^{(i-1)/2} c_{i,2j+1} x^{2j} \right) \quad \text{for} \quad i = 1, 3, \cdots \qquad (19)$$

then the contents of $S_{0,1}$, $S_{0,2}$, $S_{0,3}$, and $S_{0,4}$ are given in Figure 6 as time increases from 0, the exact timing as to when these c's should be fed into the PCA–W net will be seen in the appendix.

The output of the CA–7 subnet is multiplied at $\pi_{4,1}$ by an appropriate factor coming from SMCA. For example, the output of CA–7, $T_i(x)$ for $i \leq 7$ or the first segment of $T_i(x)$ for $i > 7$ is multiplied at $\pi_{4,1}$ by the unity coming from $S_{3,1}$ through $M_{4,1}$. The second, third, and fourth segments, if they exist, of $T_i(x)$ for $i > 7$, are multiplied by x^8, x^{16}, and x^{24} respectively. The partial result of each $T_i(x)$ is accumulated at

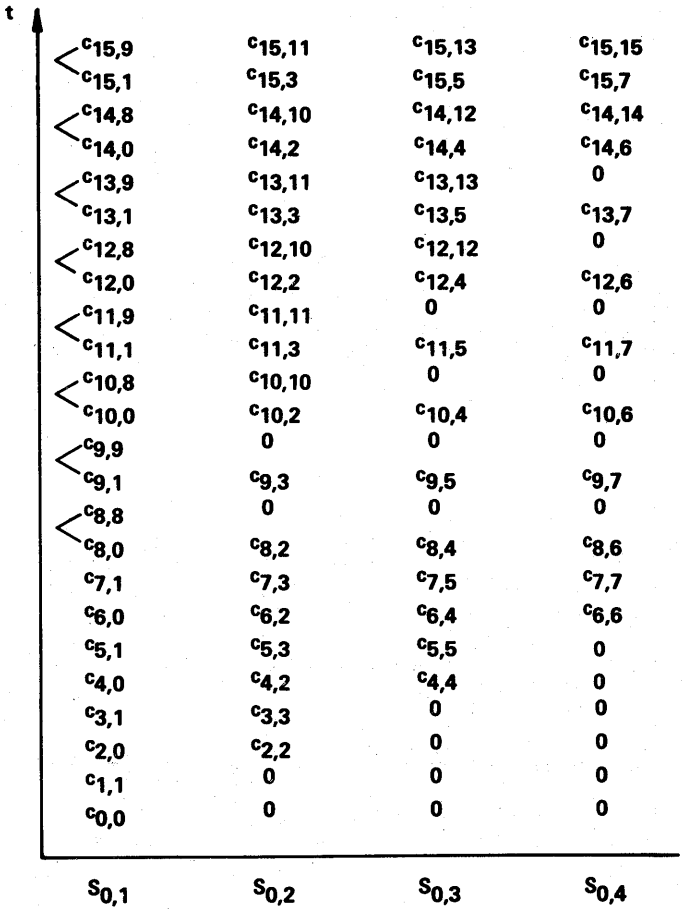


Figure 6—Contents of the initial storage layer of PCA-7 net

$\Sigma_{4,1}$ until $T_i(x)$ is completely evaluated and stored in $\Sigma_{4,1}$. $T_i(x)$ is then multiplied at $\pi_{5,1}$ by a_i. All $a_iT_i(x)$ are accumulated at $\Sigma_{5,1}$ sequentially with i increasing from 0 to the prescribed n.

It is thus seen that once the PCA net is filled up with significant data then almost every piece of the hardware is in constant use until no more inputs are fed. In this manner, the overall utilization factor is very high, hence more economical and practical than the totally combinational arithmetic net. One important unique advantage of employing Chebyshev approximation is that when changing from approximating one given function to the other, nothing of the CA net needs to be changed except a new set of coefficients a_i's should be prepared.

The H-level graph of Figure 6 is shown in Figure 7 with the assumption that the precision is no greater than 17 digits. Detailed timing analysis based on this is shown in the appendix.

Computational complexity

Speed, cost, and error studied here refer only to the H-graph of a CA net. Speed is measured by the delay through the longest information flow path of the net, which depends not only on the topology of the net but also on the precision of the data words. Cost, defined as the count of ABB's is a function of the degrees of polynomials and the series as well as the precision of the data word. Error in a net comes only from round-off operations if all input data are assumed free from inherent errors. We will use precision and the degree of polynomial and series as independent variables in the following study. Further, we assume $r = 16$, minimal-redundancy, and precision $p \leq 17$. The results only show the upper bounds of cost, speed, and error.

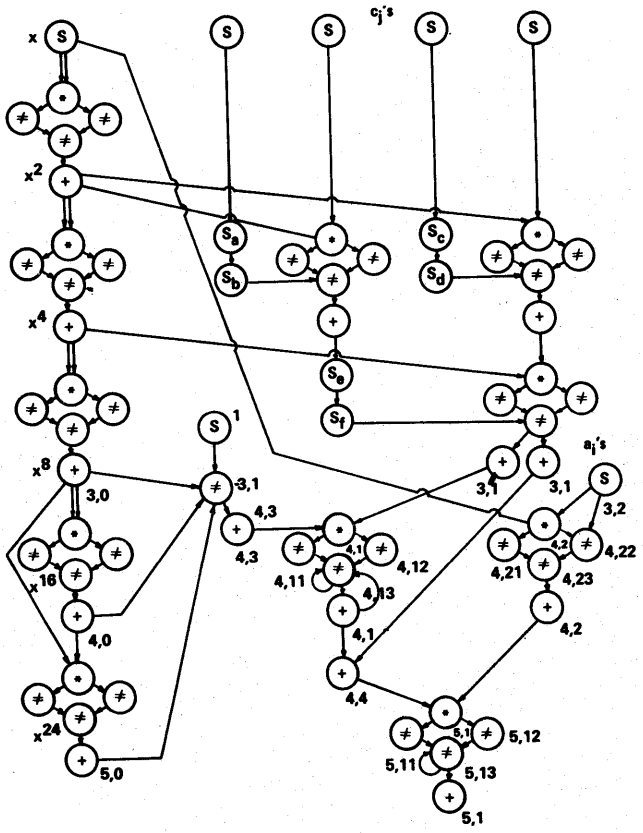


Figure 7—PCA-7 net for evaluating Chebyshev series—H-level

Delay analysis

With the detailed timing analysis, we are able to construct tables, e.g., Tables (1) and (2), showing the time required to evaluate a given Chebyshev series on PCA nets of different widths. For each width we consider two cases: one for evaluating full Chebyshev series; the other for evaluating even Chebyshev series in which odd terms are missing. In the tables t_0 is the time at which the data for a given segment are in the first storage layer of the PCA net and t_f is the time at which the segment is evaluated and stored in the last vertex. In each table it is implied that the independent variable x is fed into the PCA net at $t = 0$.

Among the functions whose Chebyshev approximation tables are available in Reference 6, the following functions are approximated by full Chebyshev series: Exponential, Logarithmic, Gamma, Exponential Integral, some Bessel functions, and the following functions are approximated by even Chebyshev series: Trigonometric, Inverse Trigonometric, Inverse Hyperbolic, Error, and some Bessel functions.

The relationship shown in Tables 1 and 2 can be characterized by general formula which will be derived below.

(a) For full Chebyshev series,

$$f(x) = \sum_{i=0}^{n} a_i T_i(x) \qquad i = 0, 1, 2, 3, \cdots, n-1, n$$

let $g = \lfloor i/w + 1 \rfloor$,

$$h = i - g(w+1)$$

then

$$t_{0,i} = 4 + \left(\sum_{j=1}^{g} j\right)(w+1) + (g+1)h$$

$$t_{f,i} = t_{0,i} + g + [4(\lfloor \log_2 w \rfloor + 1) + 5] \quad (20)$$

TABLE I—Delay Analysis of PCA-7 Net—Full Chebyshev Series

i		0	1	2	3	4	5	6	7		8	9	10	11	12	13	14	15		16	17	18	19	20	31	22	23		24	25	26
$t_{0,i}$	$T_{i,0}$	4	5	6	7	8	9	10	11	$T_{i,0}$	12	14	16	18	20	22	24	26	$T_{i,0}$	28	31	34	37	40	43	46	49	$T_{i,0}$	52	56	60
										$T_{i,1}$	13	15	17	19	21	23	25	27	$T_{i,1}$	29	32	35	38	41	44	47	50	$T_{i,1}$	53	57	61
																			$T_{i,2}$	30	33	36	39	42	45	48	51	$T_{i,2}$	54	58	62
																												$T_{i,3}$	55	59	63
$t_{f,i}$		17	18	19	20	21	22	23	24		30	32	34	36	38	40	42	44		47	50	53	56	59	62	65	68		72	76	80

For instance, $a_{19}T_{19}(x)$ in PCA–7 net, given

$$w = 7, \qquad i = 19$$

then

$$g = \lfloor 19/8 \rfloor = 2$$

$$h = 19 - 2(7 + 1) = 3$$

$$t_{0,19} = 4 + \left(\sum_{r=1}^{2} r\right)(7 + 1) + (2 + 1)3$$

$$= 4 + 24 + 9 = 37$$

$$t_{f,19} = 37 + [4(\lfloor \log_2 7 \rfloor) + 1] + 5 + 2 = 56$$

(b) For even Chebyshev series

$$f(x) = \sum_{i=0}^{n} a_i T_i(x) \qquad i = 0, 2, 4, \cdots, n-2, n$$

let

$$g = \lfloor i/w + 1 \rfloor$$

$$h' = [1 - g(w + 1)]/2 \qquad (21)$$

then

$$t_{0,i}' = 4 + \left(\sum_{j=1}^{g} j\right)(w + 1)/2 + (g + 1)h'$$

$$t_{f,i}' = t_{0,i}' + g + [4(\lfloor \log_2 w \rfloor + 1) + 5] + g$$

Let TCA be the CA net which is large enough such that all T_i's required for evaluating in given Chebyshev series can be completed concurrently. As a comparison, the time needed to evaluate the Chebyshev series on this TCA net is

$$t_{\text{TCA}} = 4(\lfloor \log_2 n \rfloor + 2) \qquad (22)$$

t_{TCA} and $t_{f,n}$, as well as $t_{f,n}'$ are drawn in Figures 8 and 9.

Cost Estimate

Cost of a PCA net is reflected as a count of ABB's. The exact cost of a CA net is very difficult to obtain without the detailed knowledge of individual problems at hand. We are interested in the order of magnitude of the cost, hence, the analysis is done here without

(a) possible minimization of the H-level net;

(b) possible reduction of ABB's from a detailed study of the exact data format. Rather, we assume the precision is p everywhere. As a result, the cost of a * vertex is

$$(1 + 2 + 3 + \cdots + p + (p - 1) + (p - 2) + (p - 3)) = (p^2 + 7p - 12)/2,$$

TABLE II—Delay Analysis of PCA-7 Net—Even Chebyshev Series

i		0	2	4	6		8	10	12	14		16	18	20	22		24	26
$t_{0,i}$	$T_{i,0}$	4	5	6	7	$T_{i,0}$	8	10	12	14	$T_{i,0}$	16	19	22	25	$T_{i,0}$	28	32
						$T_{i,1}$	9	11	13	15	$T_{i,1}$	17	20	23	26	$T_{i,1}$	29	33
											$T_{i,2}$	18	21	24	27	$T_{i,2}$	30	34
																$T_{i,3}$	31	35
$t_{f,i}$		17	18	19	20		26	28	30	32		35	38	41	44		48	52

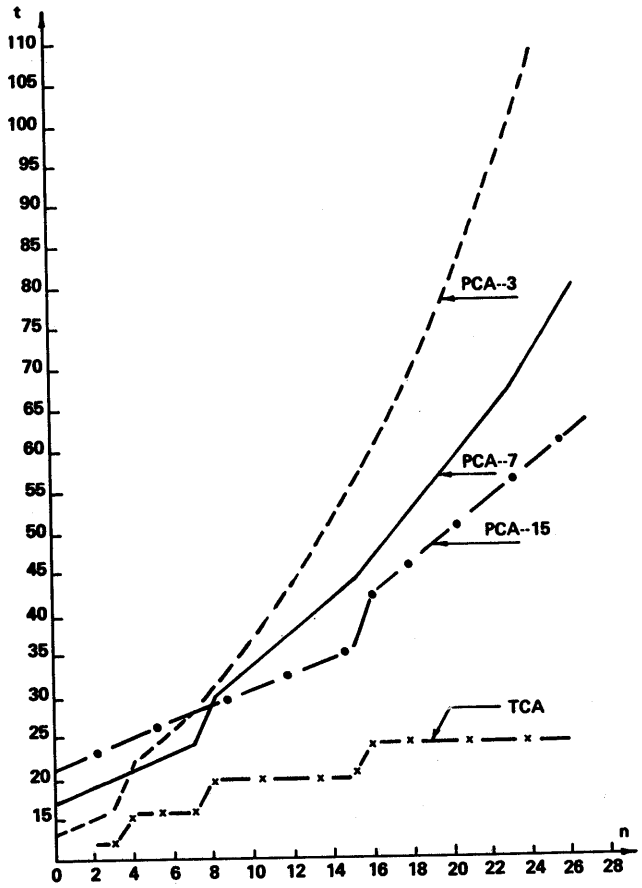


Figure 8—Delay of evaluating full Chebyshev series on a PCA-W net and on a TCA net

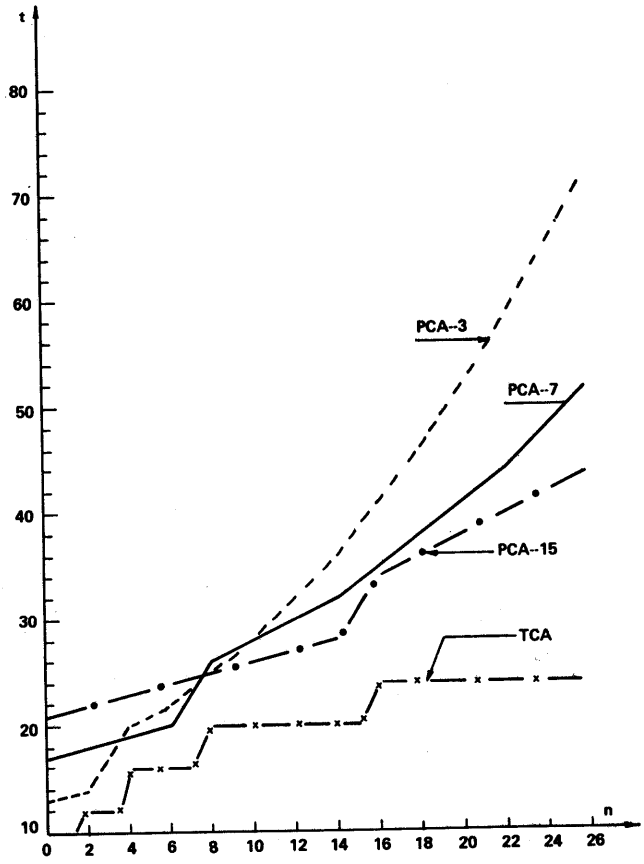


Figure 9—Delay of evaluating even Chebyshev series on a PCA-W net and on a TCA net

and that of a $\neq$, +, or S vertex is p ABB's.[5] The estimate of cost of several PCA–W nets, evaluating Chebyshev series up to the degree of n, is shown below.

PCA–7 net:

Number of π–Σ pairs:

$$N(\pi\text{–}\Sigma) = N_{\text{CA–7}}(\pi\text{–}\Sigma) + N_{\text{SMCA}}(\pi\text{–}\Sigma) + N_{\text{PSCA}}(\pi\text{–}\Sigma)$$

$$= 5 + \left\lfloor \frac{n}{8} \right\rfloor + 3$$

$$= 8 + \left\lfloor \frac{n}{8} \right\rfloor$$

Number of S vertices:

$$N(\text{S}) = N_{\text{CA–7}}(\text{S}) + N_{\text{SMCA}}(\text{S}) + N_{\text{PSCA}}(\text{S})$$

$$= 11 + 1 + 1$$

$$= 13$$

Misc.

$$1 \neq, \qquad 3 + \text{'s}$$

Cost of a π–Σ pair:

$$\$(\pi\text{–}\Sigma) = \$(*) + 3 \cdot \$(\neq) + \$(+)$$

$$= (p^2 + 15p - 12)/2$$

So the cost of a PCA–7 net is:

$$\$(\text{PCA–7}) = \left[\left(8 + \left\lfloor \frac{n}{8} \right\rfloor\right)(p^2 + 15p - 12)/2\right] + 17p$$

The general formulas of obtaining the cost of a PCA–W net, evaluating Chebyshev series up to the degree of n, are as follows:

Assume

$$w = 2^i - 1, \qquad i = 2, 3, 4, \cdots$$

then

$$N_{CA-W}(\pi\text{-}\Sigma) = \lfloor \log_2 w \rfloor + \lfloor w/2 \rfloor$$

$$N_{SMCA}(\pi\text{-}\Sigma) = \lfloor n/(w+1) \rfloor$$

$$N_{PSCA}(\pi\text{-}\Sigma) = 3$$

$$N_{CA-W}(S) = (w+1)/2 + 1 + 2[2^{\log_2(w+1)-1} - 1]$$

$$N_{SMCA}(S) = 1$$

$$N_{PSCA}(S) = 1$$

Misc.

$1 \neq$, (if $N_{SMCA}(\pi\text{-}\Sigma) \leq 16$)

$3 + 's$.

where the third term in $N_{CA-W}(S)$ accounts for the dummy storage vertices, for example, S_a, S_b, $\cdots$, S_f in Figure 7.

Thus the cost of PCA–W net is

$$\begin{aligned}\$(\text{PCA}\text{–}W) &= [(p^2 + 15p - 12)/2]N(\pi\text{-}\Sigma) \\ &\quad + p[N(S) + 4] \\ &= [(p^2 + 15p - 12)/2][\lfloor \log_2 w \rfloor \\ &\quad + \lfloor w/2 \rfloor + \lfloor n/(w+1) \rfloor + 3] \\ &\quad + p\left[\frac{w+1}{2} + 7 + 2(2^{\log 2^{(w+1)-1}} - 1)\right] \quad (23)\end{aligned}$$

Figure 10 illustrates the costs of several PCA–W nets versus the degree n of Chebyshev series with $p = 10$ (approximately 40-bit precision) and $p = 15$ (approximately 60-bit precision).

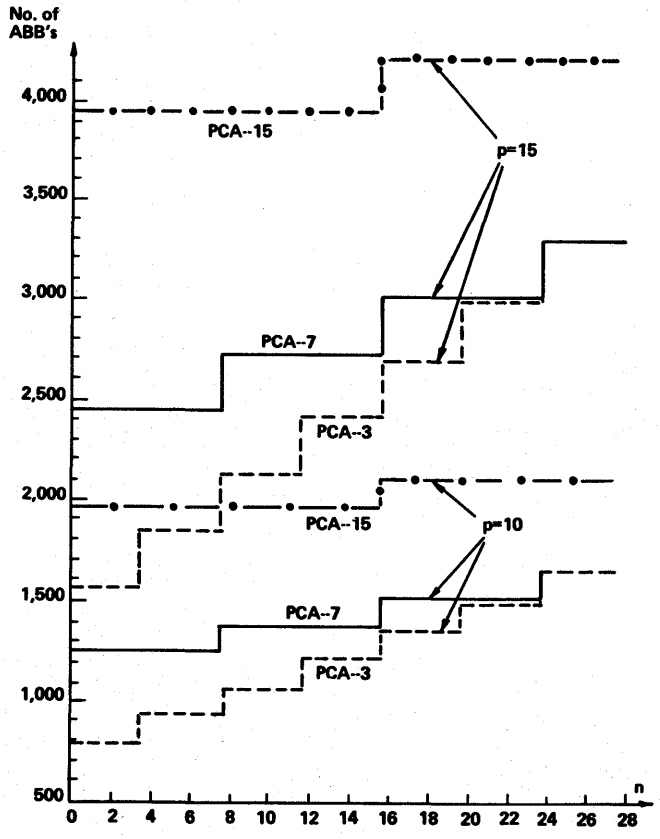


Figure 10—Cost estimate of PCA-W nets

Error analysis

As stated previously, a function $f(x)$ which is continuous and of bounded variation in $[-1, +1]$ can be expressed in terms of the Chebyshev series

$$f(x) = \sum_{i=0}^{\infty}{}' a_i T_i(x) \quad (24)$$

A practical implementation of this formula on a PCA–W net gives rise to two kinds of computational errors.

(a) Truncation error:

Due to the natural requirement that any computational job must be finished or terminated in a finite amount of time or a finite number of steps, the upper limit of the running index in Equation (24) must be replaced by a practical integer, n. That is, $f(x)$ is approximated by a truncated formula

$$f(x) \doteq \sum_{i=0}^{n} a_i T_i(x) \quad (25)$$

with a tolerable error. The truncation error thus incurred is therefore

$$e_T = \sum_{i=n+1}^{\infty} |\, a_i T_i(x) \,| \leq \sum_{i=n+1}^{\infty} |\, a_i \,| \quad (26)$$

because the magnitude of $T_i(x)$ is bounded by unity. As a result of the rapid convergence of Chebyshev series, usually a_{n+1} suffices to account for the truncation error indicated in Equation (26).

(b) Round-off errors:

Any practical computational mechanism can have only finite word lengths. Because of this inevitable limitation, the length of the results of arithmetic operations cannot increase without limit; it must be eventually rounded off. This is, of course, true in the PCA net. In addition, the values of the coefficients of Chebyshev series, a_i's, are not necessarily exact, hence

they must be chopped (rounding in S–D numbers is unbiased) to fit the prescribed word length. As a result, we introduce a new source of round-off error—an inherent error of the incoming data as far as the PCA net is concerned. It should be pointed out that the coefficients of Chebyshev polynomials, c_j's, are integers and so there is no error in representing them.

Let the accumulated round-off error in evaluating $T_i(x)$ be e_{T_i} and the round-off error of a_i be e_{a_i}, then the total round-off error in evaluating

$$\sum_{i=0}^{n} a_i T_i(x)$$

is

$$e_R = \sum_{i=0}^{n}\left[(e_{T_i})(a_i x) + \left(\frac{T_i(x)}{x}\right)(e_{a_i} x)\right] \quad \text{for odd } i$$

$$+ \sum_{i=0}^{n}\left[(e_{T_i})(a_i) + (T_i(x))(e_{a_i})\right] \quad \text{for even } i$$

$$\leq \sum_{i=0}^{n}\left[e_{T_i} a_i + e_{a_i}\right] \quad \text{for all } i \qquad (27)$$

Signed-digit number representations are symmetric with respect to zero which has a unique representation. The round-off process is simply done by chopping the data word at a certain position. Statistically, the round-off process is not biased. For minimum redundancy even radix S–D system, the upper bound of round-off error is derived as follows:

Let chopping occur at p digits after the radix point—all digits to the right are omitted. The maximum absolute value of each digit is $(r/2 + 1)^2$ and thus the upper bound of round-off error, e_r, is

$$e_r < \left(\frac{r}{2} + 1\right) r^{-(p+1)} + \left(\frac{r}{2} + 1\right) + r^{-(p+2)} + \cdots$$

$$= \left(\frac{r}{2} + 1\right) \cdot r^{-(p+1)} \cdot \frac{r}{r-1}$$

$$= \frac{r+2}{2(r-1)} \cdot r^{-p} \qquad (28)$$

For $r = 16$,

$$e_r < \tfrac{3}{5} r^{-p}$$

CONCLUSIONS

A study of existing numerical approximation techniques indicates that Chebyshev approximation offers certain unique advantages over other methods insofar as implementation on combinational arithmetic nets are concerned. Thus, a pipelined CA net has been constructed, based on a compromise of speed and cost, to evaluate Chebyshev series. This can best be summarized in Figure 11.

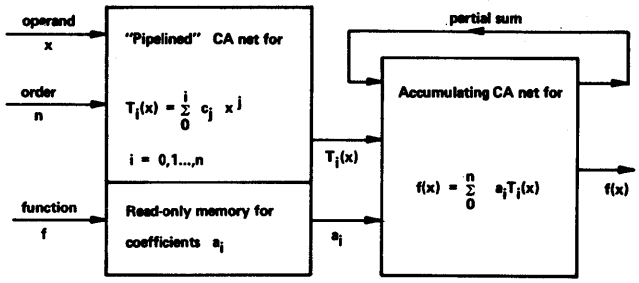


Figure 11—PCA net for Chebyshev approximation of functions

In such a system, accuracy can be improved by increasing order n at the sacrifice of speed, but without altering the internal configuration. To approximate a different function requires the change of Chebyshev coefficients a_i's only. Certain aspects of computational complexity have been investigated. Many difficult and challenging problems, such as scaling, still remain to be studied.

REFERENCES

1 S F ANDERSON T G EARLE
R E GOLDSCHMIDT D M POWERS
The IBM system/360 model 91: Floating-point execution unit
IBM Journal of Research and Development Vol 11 No 1 pp 34–53 January 1967

2 A AVIZIENIS
Signed-digit number representations for fast parallel arithmetic
IRE Trans on Electronic Computers Vol EC-10 pp 389–400 September 1961

3 A AVIZIENIS
Arithmetic microsystems for the synthesis of function generators
Proc IEEE Vol 54 pp 1910–1919 December 1966

4 A AVIZIENIS C TUNG
Design of combinational arithmetic nets
Digest First IEEE Computer Conference pp 25–28 September 1967

5 C TUNG
A combinational arithmetic function generation system
Report No 68-29 Department of Engineering UCLA June 1968 Ph.D Dissertation

6 C W CLENSHAW
Mathematical tables, Vol 5, Chebyshev series for mathematical functions
National Physics Laboratory London 1962

7 G ESTRIN
Organization of computer systems—The fixed plus variable structure computer
Proc WJCC pp 33–40 1960

8 E W CHENY
Introduction to approximation theory
McGraw-Hill 1966
9 C LANCZOS
Applied analysis
Prentice Hall 1956
10 A RALSTON H S WILF
Mathematical methods for digital computers
John Wiley and Sons 1960
11 C TUNG
A division algorithm for signed-digit arithmetic
IEEE Trans on Computers Vol C-17 pp 887–889 September 1968
12 C TUNG
The application of an arithmetic building unit to a signed-digit division algorithm
IBM Research Report RJ 591 July 15 1969

APPENDIX

Timing analysis of PCA–W nets

Timing sequence on a PCA–*W* net must be carefully studied not only to ensure a correct evaluation of the Chebyshev but also to shorten the delay through the net as well as to minimize the time spacing between consecutive segments fed into the PCA net. A PCA–*W* net expressed in the hardware level lends itself more readily to timing analysis. It should be pointed out that an *H*-level vertex consists of a row of ABB's. Each ABB is equipped with output gate and clock for regulating or controlling information flow in a CA net.[5] Figure 7 is the hardware level net of Figure 5 in which the precision is allowed up to 17 (radix 16) significant digits. The timing analysis considers two cases separately: $T_i(x)$ with $i \leq 7$ and $T_i(x)$ with $i > 7$. It should be emphasized that in the following discussion, for all *H*-level vertices, the output gates are always open and clocks are always effective unless otherwise stated.

When $i \leq 7$, $T_i(x)$ needs not be segmented. Assume at $t = t_0$, x is fed to $S_{0,0}$ and at $t = t_0 + 4$ all coefficients c_j of $T_i(x)$ are in the first storage layer ($S_{0,1}, \cdots, S_{0,4}$). At $t = t_0 + 12$, $T_i(x)$ is ready at $+_{3,1}$ and $+_{3,1}'$, the output of the CA–7 subnet. $T_i(x)$ must be multiplied with an a_i at $t = t_0 + 14$, hence at $t = t_0 + 11$ the gate of $*_{4,2}$ is closed to produce a_i. Only at $t = t_0 + 16$, the clock of $\neq_{5,13}$ is effective such that the partial result, $a_iT_i(x)$, can be registered there. Since there is no extra delay needed while accumulating the result in $\neq_{5,13}$, $T_{i+1}(x)$ can come right after $T_i(x)$. According to the criterion outlined in the text the minimum delay for evaluating

$$f(x) = \sum_{i=0}^{n} a_iT_i(x) \qquad n \leq 7 \qquad \text{(A-1)}$$

with a given PCA–*W* net is

$$T = n\Delta t + \Delta T \qquad \text{(A-2)}$$

if ΔT is proved to be the shortest possible delay through the net for computing and solving one segment of the given function. Referring to Figure 7, we see ΔT is 17 time units, while the shortest possible delay is 16 time units (four multiplications have to take place in series). The extra delay is caused by $+_{4,4}$. The passage through $+_{4,4}$ seems redundant but it is actually indispensable as will be seen in the following discussion. Therefore, the current design requires minimum amount of time to evaluate Equation A-1. Meantime, the gates of $S_{3,1}$, $+_{3,0}$, $+_{4,0}$, $+_{5,0}$, and $+_{4,11}$ are closed because they are not needed for computing $T_i(x)$ with $i \leq 7$.

When $i > 7$, $T_i(x)$ must be segmented as shown in Figure 6. Let us denote the segments of $T_i(x)$ by $T_{i,0}(x)$, $T_{i,1}(x)$, $\cdots$, $T_{i,k}(x)$. In order to minimize the overall delay through the net the input data for $T_{i,k}(x)$ must be fed into the net immediately after the input data for $T_{i,k-1}(x)$. Whether this requirement can be met depends on a subtle arrangement of vertices $*_{4,1}$, $\neq_{4,11}$, $\neq_{4,12}$, $\neq_{4,13}$, $+_{4,1}$, and $+_{4,4}$.

$T_8(x)$ is broken into $T_{8,0}(x)$ and $T_{8,1}(x)$. With the gate of $+_{3,1}$ closed from now on, $T_{8,0}(x)$ and $T_{8,1}(x)$ will be ready in $+_{3,1}'$ at $t = t_0 + 12$ and $t = t_0 + 13$ respectively. Only at $t = t_0 + 11$ and $t = t_0 + 12$ the gates of $S_{3,1}$ and $+_{3,0}$ will be open respectively to insure proper multiplication of $1 \cdot T_{8,0}(x)$ and $x^8 \cdot T_{8,1}(x)$ in $*_{4,1}$ at $t = t_0 + 13$ and $t = t_0 + 14$. At $t = t_0 + 16$, $T_8(x)$ is formed in $\neq_{4,13}$. At $t = t_0 + 17$, $\neq_{4,13}$ holds the sum of $T_8(x)$ and $T_{9,0}(x)/x$ with the gate of $+_{4,1}$ still closed. At $t = t_0 + 18$, the gate of $+_{4,1}$ is permitted to open for one time unit such that $T_8(x)$ would be in $+_{4,4}$. At the same time, $\neq_{4,13}$ holding the sum of $T_8(x)$ and $T_{9,0}(x)/x$ receives $T_{9,1}(x)/x$ from $\neq_{4,11}$ and $\neq_{4,12}$ and the complemented value of $T_8(x)$ from $+_{4,1}$. The net effect at $\neq_{4,13}$ is to produce $T_9(x)/x$ at $t = t_0 + 18$.

Assume at $t = t_1$, $T_{16,0}(x)$, $T_{16,1}(x)$ and $T_{16,2}(x)$ are in $\neq_{4,11}$ (and $\neq_{4,12}$), $*_{4,1}$, and $+_{3,1}'$ respectively, while the contents of $\neq_{4,13}$ is $T_{15}(x)/x$. The gates of $S_{3,1}$, $+_{3,0}$, and $+_{4,0}$ are permitted to open at $t = t_1 - 3$, $t_1 - 2$, $t_1 - 1$, respectively. At $t = t_1 + 1$, $\neq_{4,13}$ holds the sum of $T_{15}(x)/x$ and $1 \cdot T_{16,0}(x)$ and $+_{4,1}$ holds $T_{15}(x)/x$ with its gate still closed. At $t = t_1 + 2$, the gate of $+_{4,1}$ is open, $\neq_{4,13}$ now holds $T_{15}(x)/x + T_{16,0}(x) + x^8T_{16,1}(x) - T_{15}(x)/x = T_{16,0}(x) + x^8T_{16,1}(x)$. With the gate of $+_{4,1}$ closed again at $t = t_1 + 3$, the contents of $\neq_{4,13}$ become $T_{16,0}(x) + x^8T_{16,1}(x) + x^{16}T_{16,2}(x) = T_{16}(x)$. The transition from 3-segment $T_i(x)$ to 4-segment $T_i(x)$ and the transition from 4 to 5 and so forth are done in the same manner. The preparation of $a_i(a_ix)$, the multiplication of a_i and $T_i(x)$ (a_ix and $T_i(x)/x$) at $*_{5,1}$, and the registra-

tion and accumulation at $\neq_{5,13}$ are done exactly in the same way as they were in the case of $a_iT_i(x)$ for $i \leq 7$. The existence of $+_{4,4}$ is necessary because one of the inputs of $*_{5,1}$ must come from either $+_{3,1}$ or $+_{4,1}$ and $+_{4,4}$ must be there to serve as a selector. Since input data for all segments of $T_i(x)$ can be fed into the net one immediately after the other as we expected, and ΔT is also held minimum, the Chebyshev series is evaluated on this PCA–7 net at the fastest possible pace, according to Equation (13).

To insure proper and reliable operations in the PCA–7 net shown in Figure 7, the gates and clocks of

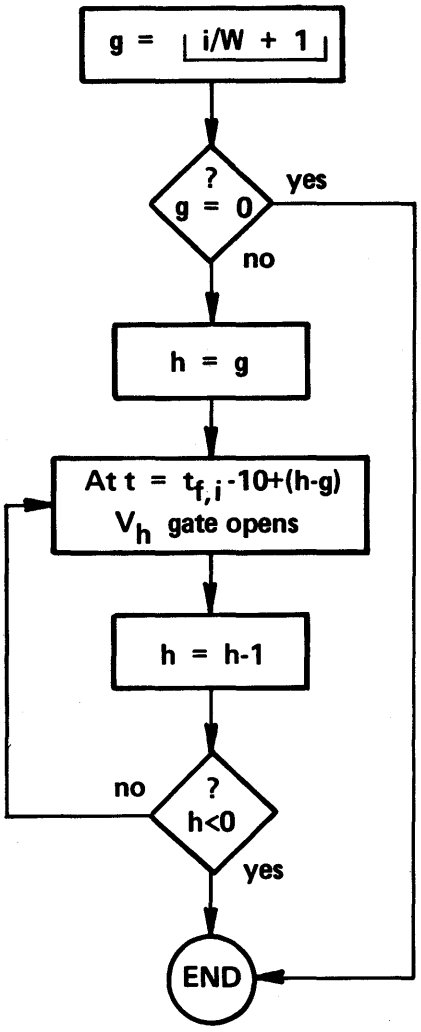


Figure A-1—Determination of gate conditions in SMCA subnet

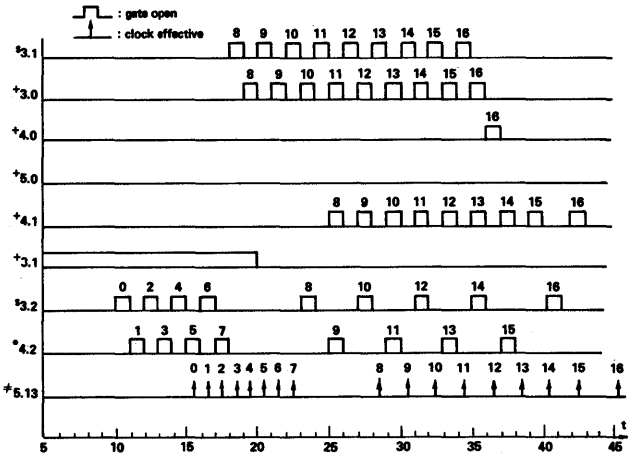


Figure A-2—Control timing chart for Figure 7

the following H-level vertices (or, equivalently, ABB's).

$$S_{3,1},\ +_{3,0},\ +_{4,0},\ +_{5,0},\ +_{4,1},\ +_{3,1},\ S_{3,2},\ \neq_{4,2},\ \neq_{5,13}$$

must be carefully studied while for the remaining H-level vertices, the following assumption holds, "Output gates are always open and clocks are always effective."

Let V_h be the H-level vertex in the SMCA subnet where $x^{h(w+1)}$ is formed, then in our example (Figure 7)

$$V_0\colon\ S_{3,1},\ V_1\colon\ +_{3,0},\ V_2\colon\ +_{4,0},\ V_3\colon\ +_{5,0}.$$

Their clocks are always effective; gates open only at certain instants, which are defined by the flow-chart shown in Figure A-1, in which $t_{f,i}$ refers to the instant at which $a_iT_i(x)$ is in the last H-level vertex of the net, i.e., $+_{5,1}$ in Figure 7.

For $+_{4,1}$, clocks are always effective while its gate opens only at $t = t_{f,i} - 4$. The gate of $+_{3,1}$ opens only for the first $w + 4(\lfloor \log_2 w \rfloor + 1) + 1$ time units with the clocks always activated. The gate of $\neq_{5,13}$ always opens while clocks are activated only at $t = t_{f,i} - 1$. At $t = t_{f,i} - 6$, the gate of $S_{3,2}(*_{4,2})$ opens (closes) for even Chevyshev polynomial and the gate of $*_{4,2}(S_{3,2})$ opens (closes) for odd Chebyshev polynomial.

With the knowledge obtained so far, we draw a control timing chart in Figure A-2 for evaluating

$$\sum_{i=0}^{16} a_iT_i(x)$$

on a PCA–7 net.

Operating systems architecture

by HARRY KATZAN, JR.

Pratt Institute
Brooklyn, New York

INTRODUCTION

Operating systems architecture refers to the overall design of hardware and software components and their operational effectiveness as a whole. To be effective, however, an operating system must not only be cognizant of the collection of hardware and software modules, but must also be designed in light of the programs and data which the system processes and the people which it serves. The absence of formal theory on operating systems and the lack of standard terminology have caused much confusion among users. The problem is particularly apparent when comparing systems where the same terms are applied to a variety of concepts and levels of implementation.

The purposes of this paper are threefold: (1) to present the basic properties with which operating systems can be grouped, classified, and evaluated; (2) to identify the major categories into which operating systems can be classed and to give concrete examples of each; and (3) to discuss resource management in operating systems with an emphasis on storage allocation and processor scheduling.

First, seven properties, used to classify operating systems, are briefly described. Then, the major categories into which operating systems can be classed are given and their most significant attributes are noted. Lastly, the most significant factors in operating systems design, i.e., storage allocation and processor scheduling, are treated in detail.

PROPERTIES OF OPERATING SYSTEMS

An operating system tends to be classified, informally, on the basis of how the facilities of the system are managed or allocated to the user. Accordingly, the number of properties, or combinations of properties, which contribute to the classification, is very large. Seven of these properties dominate the remainder and are introduced in the following paragraphs.

Access

Access is concerned with how the user interacts with the system. Does he access the system via a remote terminal or does he submit his work in a batch processing environment? If the user is at a remote location, what is the nature of his terminal device? Is it a RJE/RJO work station or is it a keyboard or CRT type device? Is a command system available so that the user can enter into a dialogue with the operating system? Can the user initiate batch tasks or query the status of them when at a remote terminal? Does the facility exist for conversing with a problem program from a terminal device?

Utilization

Utilization is concerned with the manner in which the system is used. Is the system *closed* so that the user is limited to a specific programming language or is the system *open* allowing the user access to all of the system facilities? How must the user structure his programs—planned overlay, dynamic segmentation, single-level store? Can the user prepare and debug programs on line or is he limited to querying the system? What facilities are available for data editing and retrieval? In the area of data management, what access methods, file organization, and record types are permitted? Does the data management system have provisions for using the internal and external storage management facilities of the system? Lastly, what execution-time options are permitted by the operating system at run time as compared to compile time?

Performance

Performance deals with the quality of service to the installation and to the user. Does the operating system design philosophy attempt to maximize the use of

system resources, maximize throughput, or guarantee a given level of terminal response? What is the probability that the system will be available to the user when needed? Does the user lose his data sets if the system fails (data set integrity)? What facilities are available for system error recovery?

Scheduling

Scheduling determines how processing time is allocated to the jobs (or tasks) which reside in some form in the system. What scheduling philosophy is used—sequential, natural wait, priority, time slicing, demand? What is the nature of the scheduling algorithm—round robin, exponential with priority queues, table driven?

Storage management

Storage management is concerned with how main storage and external storage are allocated to the users. Is main storage fragmented, divided into logical regions, allocated on a page basis or are programs swapped? Is external storage allocated in fixed-size increments or on a demand basis with secondary allocations, as required?

Sharing

Sharing is the functional capability of the system to share programs, data, and hardware devices. Does the system permit readonly and re-entrant code so that programs can be shared during execution? Can data sets be shared without duplicating them? At what level of access? Can hardware devices be shared among users giving each the illusion that he has a logical device to himself?

Configuration management

Configuration management is concerned with the real physical system and the logical system as seen by the user. Physically, how is the system organized and how can this organization be varied? Does the capability exist of partitioning off a maintenance subsystem? Similarly, can a failing CPU, core box, channel, or I/O device be removed from the system? Logically, does the user have a machine to himself, a large virtual memory, a fixed partition?

Obviously, the properties are not exhaustive in the sense that all operating systems, real or hypothetical, can be automatically classified. The properties do form a basis for comparison and are used in the next section to identify the major categories of operating systems.

CATEGORIES OF OPERATING SYSTEMS

An *operating system*[1] is an integrated set of control programs and processing programs designed to maximize the overall operating effectiveness of a computer system. Early operating systems increased system performance by simplifying the operations side of the system. Current operating systems additionally attempt to maximize the use of hardware resources while maintaining a high level of work throughput or providing a certain level of terminal response. A multitude of programmer services are usually provided, as well.

Multiprogramming

A *multiprogramming system*[2,3] is an operating system designed to maintain a high level of work throughput while maximizing the use of hardware resources. As each job enters the system, an internal priority, which is a function of external priority and arrival sequence, is developed. This internal priority is used for processor scheduling. During multiprogramming operation, the program with the highest internal priority runs until a natural wait is encountered. While this wait is being serviced, processor control is turned over to the program with the next highest priority until the first program's wait is satisfied, at which time, processor control is returned to the high priority program, regardless if the second program can still make use of the system. The first job has, in a sense, demanded control of the system. The concept is usually extended to several levels and is termed the *level of multiprogramming.* A multiprogramming system is characterized by: (1) Limited *access* traditionally limited to tape or card SYSIN and printer or tape SYSOUT. RJE/RJO may be implemented but on-line real-time processing usually requires a specially written problem program. (2) *Utilization* is most frequently restricted to batch type operations with data management facilities being provided by the system. Planned overlay is usually required for large programs with most debugging being done off line. (3) *Performance* is oriented towards high throughput and maximum utilization of hardware resources. A given level of service is not guaranteed and the processing of jobs is determined by operational procedures. (4) *Scheduling* of work usually involves priority, natural wait, and demand techniques. In some systems, a unit of work may spawn other units of work providing parallel processing, to some degree. (5) *Storage management* techniques vary between sys-

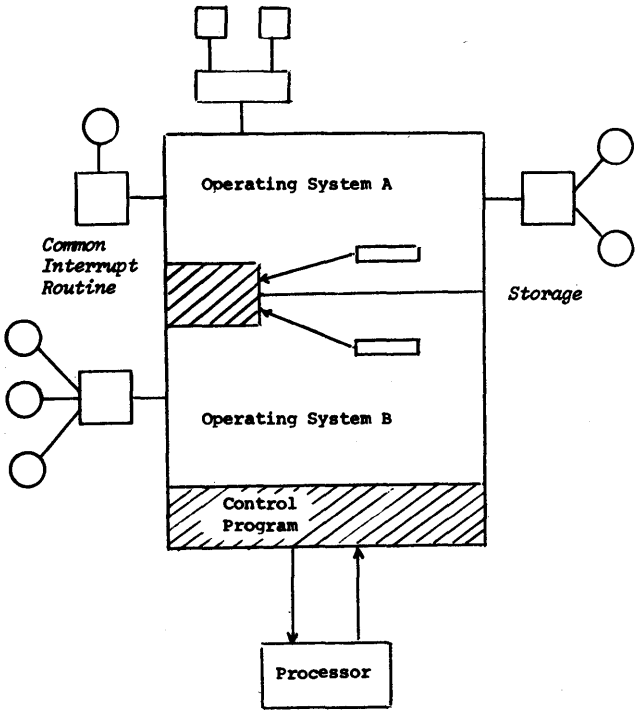


Figure 1—Hypervisor multiprogramming

tems with most systems using a fixed partition size or a logical region for problem programs. Paging techniques with dynamic address translation have been used with some success. (6) *Sharing* of system routines is frequently provided while the sharing of problem programs is a rarity. When a central catalog of data set information is provided, data set sharing, at various levels of access, is available. Otherwise, data set sharing is accomplished on an ad hoc basis. (7) *Configuration management* is usually limited to the existing physical system with the users being given a portion, fixed or variable, of actual storage.

Hypervisor multiprogramming

One of the problems frequently faced by installation management involves running two different operating systems, each of which requires a dedicated but identical machine. A *hypervisor* is a control program that, along with a special hardware feature, permits two operating systems to share a common computing system. A relatively small hypervisor control program (see Figure 1) is required which interfaces the two systems. Although only one processor is involved, a hardware prefix register divides storage into two logically separate memories, each of which is utilized by an operating system. I/O channels and devices are dedicated to one or the other operating system and use the hardware prefix register to know to which half of storage to go. All interrupts are indirectly routed to a *common interrupt routine* which decides which operating system should receive the most recent interrupt. Processor control is then passed to the hypervisor control program for dispatching. The hypervisor control program loads the prefix register and usually dispatches processor control to the operating system which received the last interrupt. Alternate dispatching rules are to give one operating system priority over another or to give one operating system control of the processor after a fixed number of interrupts have been received by the other side. Hypervisors are particularly useful when it is necessary to run an emulator and an operating system at the same time. Similar to multiprogramming systems, a hypervisor is characterized by: (1) limited access; (2) batch utilization; (3) high throughput performance; (4) priority, natural wait, and demand scheduling; (5) basic storage management techniques; (6) limited sharing facilities; and (7) configuration management determined by the operating systems that are run as subsystems.

Time sharing

Although time-sharing is used in a variety of contexts, it most frequently refers to the allocation of hardware resources to several users in a time dependent fashion. More specifically, a time-sharing system concurrently supports multiple remote users engaging in a series of interactions with the system to develop or debug a program, run a program, or obtain information from the system. The basic philosophy behind time sharing is to give the remote user the operational advantages of having a machine to himself by using his *think*, *reaction*, or *I/O* time to run other programs. Operation of a time sharing system is summarized as follows:*

> Time-shared operation of a computer system permits the allocation of both space and time on a temporary and dynamically changing basis. Several user programs can reside in computer storage at one time while many others reside temporarily on auxiliary storage such as disc or drum. Computer control is turned over to a resident program for a scheduled time interval or until the program reaches

* See reference [1], p. 190.

a relay point (such as an I/O operation), depending upon the priority structure and control algorithm. At this time, processor control is turned over to another program. A nonactive program may continue to reside in computer storage or may be moved to auxiliary storage, to make room for other programs, and subsequently be reloaded when its next turn for machine use occurs.

A time-sharing system is characterized by: (1) Remote *access* with keyboard or CRT devices and possibly RJE/RJO work stations. (2) Varied *utilization* ranging from a closed system such as QUIKTRAN[4], APL/360[5], or BASIC[6], to an open system such as MULTICS[7] or TSS/360[8,9]. In most open systems, a single-level store, on-line debugging facilities, and an extensive file system are also available. (3) *Performance* in most time sharing systems is mainly centered around dividing processor time among the users and providing fast response to terminal requests. Management of other resources in a time-sharing system is usually towards this end. (4) A given level of user service is maintained by giving users a short slice of processor time at frequent intervals according to a *scheduling* algorithm. The most frequently used scheduling algorithms are round robin and exponential with priority queues. (5) Varied *storage management* techniques are used depending upon the hardware and the sophistication of the software. Swapping and paging techniques have been used with great success. In the latter category, direct-access storage devices and large capacity core storage have both been used as paging devices. (6) Most open time-sharing systems permit *code sharing* during execution and language processors, data management routines, and command system programs are frequently shared. If public storage is provided, then *data sets sharing* is also available. In closed systems, the level of sharing is determined by the programming language used and its method of implementation. (7) In a utility class time-sharing system, *configuration management* facilities are required for preventative maintenance and for the repairing of faulty equipment. Multiple processors, storage units, and data channels are provided with many large time-sharing systems; thus the hardware resources are available for configuring the system to meet operational needs. In some time-sharing systems, the user has a logical machine to himself provided through a combination of hardware and software facilities. This topic is covered in the next section on virtual machines.

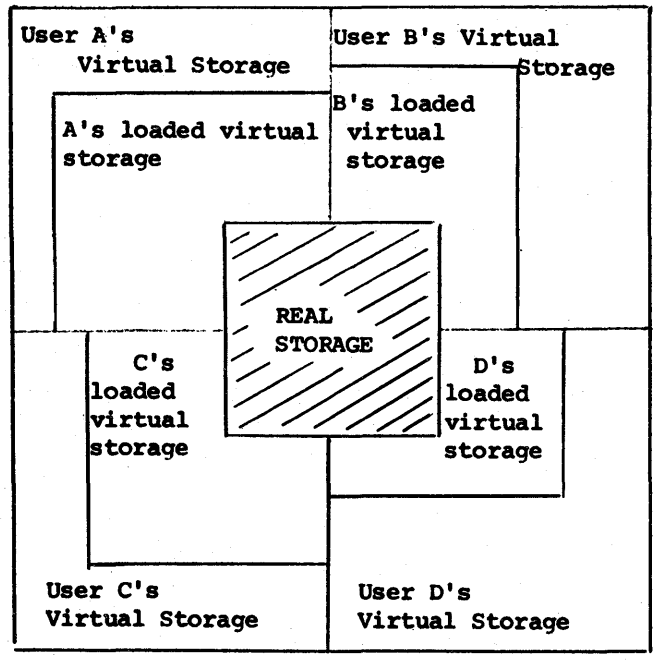


Figure 2—Virtual storage

Virtual systems

A virtual system is one which provides a logical resource which does not necessarily have a physical counterpart. *Virtual storage* systems[7,8,9,10,11] (see Figure 2) are widely known and provide the user with a large single level store achieved through a combination of hardware and software components. A virtual storage system is characterized by the fact that real storage

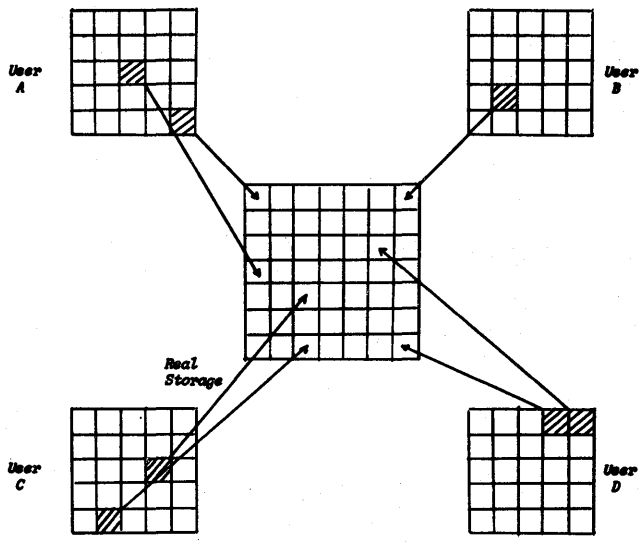


Figure 3—Loaded virtual storage

contains only that part of a user's program which *need* be there for execution to proceed. The basic philosophy of virtual storage lends itself to paging (Figure 3) and is usually associated with *dynamic address translation*, as introduced later in the paper.

A *virtual machine* is an extension to the virtual storage concept which gives the user a *logical* replica of an actual hardware system. Whereas in a virtual storage system, a user could run programs, in a virtual machine, a user or installation can run complete operating systems. In addition to using the virtual storage concept, a virtual machine system contains a control program[12] which allocates resources to the respective virtual machines and processes privileged instructions which are issued by a particular operating system.

Although virtual systems are usually associated with time-sharing, the concept is more general and applies equally well to multiprogramming systems. Virtual systems tend to be most effective in operating environments where dynamic storage allocation, dynamic program relocation, simple program structure, and scheduling algorithms are of concern. Virtual systems using fixed size pages and dynamic address translation also lend themselves to sharing and most systems using this design philosophy have implemented code sharing during execution to some extent.

Tri-level operating systems

In a conventional operating system (Figure 4), two levels of control are available, each of which corresponds to a segment of core storage. Level one contains the supervisor program and all associated routines for job control and data management. User programs, language processors, and utility programs run at level two and are regarded as problem programs by the supervisor program. Control is passed from level two to level one by hardware facilities, usually termed an *interrupt*, and level one services are able to completely sustain level two needs.

In a virtual system, another level of complexity is required. Logical as well as physical resources must be maintained and allocated. Thus, in virtual systems, allocation of resources* is relegated to the supervisor or control program and typical job control and data management routines are included as a job monitor program (Figure 5) which exists as a second level.

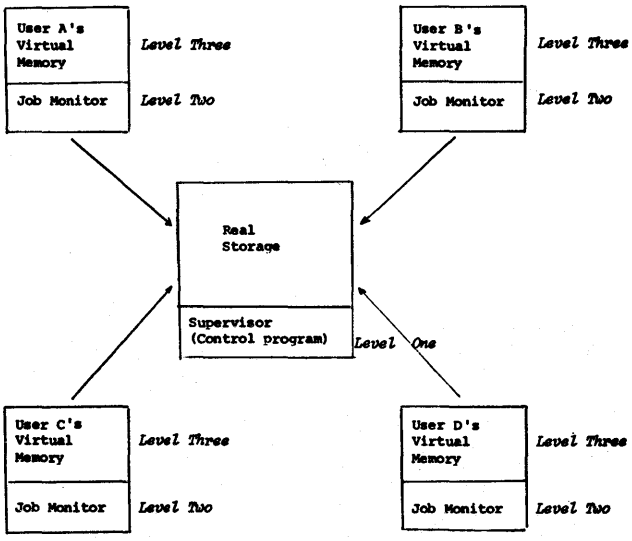


Figure 5—Tri-level operating system

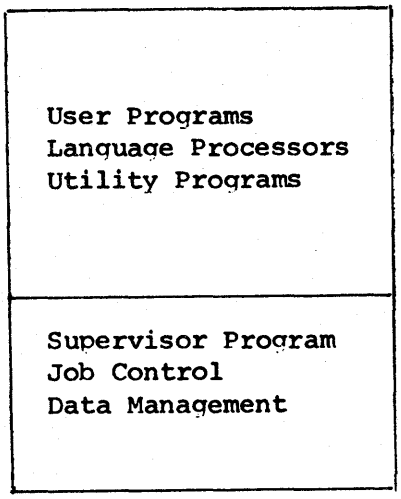


Figure 4—Conventional operating system

Processing programs then execute at a third level. The need for communication between level three and level two exists and is satisfied by a *virtual interrupt* implemented in software analogous in a real sense to the hardware interrupt discussed earlier. Level one control programs are characterized as follows: one per system, executes in the supervisor state, runs in the non-relocated mode, is not time sliced, and is core resident. Similarly, level two monitor programs are characterized by: one per user, executes in the problem state, runs in the relocated mode, is time sliced, and is pageable.

* Such as CPU time, core storage, and I/O facilities.

RESOURCE MANAGEMENT

In modern operating systems, the allocation of hardware resources among users is a major task. Two resources directly affect performance and utilization: storage management and scheduling. Both topics were introduced earlier. The most widely used implementation techniques are discussed here.

Storage management

In either a 2-level or 3-level operating system, available storage is divided into two areas: a fixed area for the supervisor program and a dynamic area for the user programs. If no multiprogramming or time sharing is done, then a user program executes serially in the dynamic area. When he has completed his use of the CPU, then the dynamic area is allocated to the next user.

When more than one user shares the dynamic area, such as in multiprogramming or time sharing, then storage management becomes a problem for which various techniques have been developed. They are arbitrarily classed as multiprogramming techniques or time-sharing techniques although the point of departure is not well-defined. *Multiprogramming techniques* include fixed partition, region allocation, and roll in/roll out. *Time-sharing techniques* include core resident, swapping, and paging.

In a *fixed partition* system, the dynamic storage area is divided into fixed sub-areas called partitions. As a job enters the system, it specifies how much storage it needs. On the basis of the space requirements specified, it is assigned to a fixed partition and must operate within that area using planned program structure whenever necessary. In a *region allocation* system, a variable number of jobs may use the system. Just before a job is initiated, a request is made to dynamically allocate enough storage to that job. Once a job is initiated, however, it is constrained to operate within that region. In a logical sense, fences are created within the dynamic area. *Roll in/roll out* is a variation of region allocation which effectively enables one job to borrow from another job if space requirements can not be fulfilled from the dynamic area. The borrowed region is rolled back in and returned to the original owner whenever he demands the CPU or when the space is no longer needed by the borrower.

The most fundamental technique for storage management in time sharing is *core resident*. In a core resident system, all active tasks are kept in main storage. This method reduces system overhead and I/O activity but is obviously limited by the size of core storage. Large capacity storage (LCS) is frequently used in a hierarchical sense with main storage and provides a cost effective means of increasing the number of potential users. Large capacity storage is sufficiently fast to satisfy the operational needs of a user at a remote terminal. *Swapping* is the most frequently used method of storage management in time sharing. At the end of a time slice, user A's program is written out to auxiliary storage and user B's is brought in for execution. All necessary control information is saved between invocations. In the above case, the system would have to wait while user B's program was brought in for execution. Thus, two or more partitions can be used for swapping to reduce the I/O wait. The use of several partitions permits other user programs to be on their way in or on their way out while one user's program is executing. This method reduces *wait* time but increases the amount of system housekeeping and overhead. A variation to the single partition approach is the *onion-skin* method used with the CTSS system at M.I.T.[13] With this method, only enough of user A's program is written out to accommodate user B. In a sense, user A's program is peeled back for user B's program. If user C requires still more space than B, then A is peeled back even more. In a *paging* system, main storage is divided into fixed-size blocks called pages. Pages are allocated to users as needed and a single user's program need not occupy consecutive pages, as implied in Figure 3. Thus a translation is required between a user's virtual storage, which is contiguous, and real storage, which is not. A technique called *dynamic address translation* is employed that uses a table look up, implemented in hardware, to perform the translation. First, the address field is segmented to permit a hierarchical set of look up tables (Figure 6). Then, each effective computer address goes through an address translation process (Figure 7) before operands are fetched from storage. The process is usually speeded up with a small associative memory (Figure 8). When a user program references a page that is not in main storage, a hardware interrupt is generated. The interrupt is fielded by the supervisor program which brings the needed page in for execution. Meanwhile, another user can use the processor. Look up tables (Figure 7) are maintained such that when a page is brought into

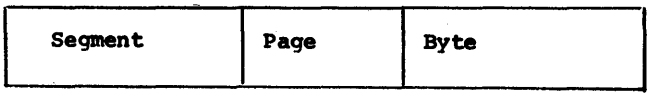


Figure 6—Segmentation

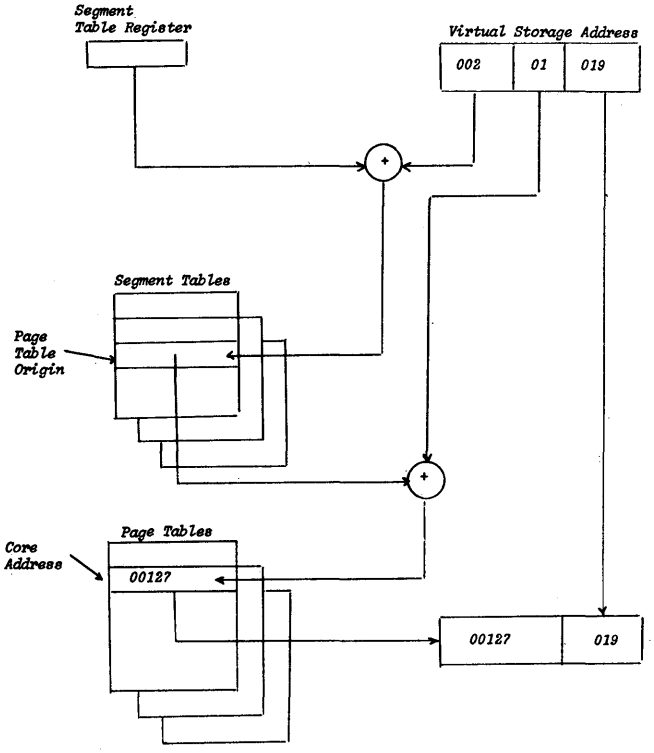


Figure 7—Dynamic address translation

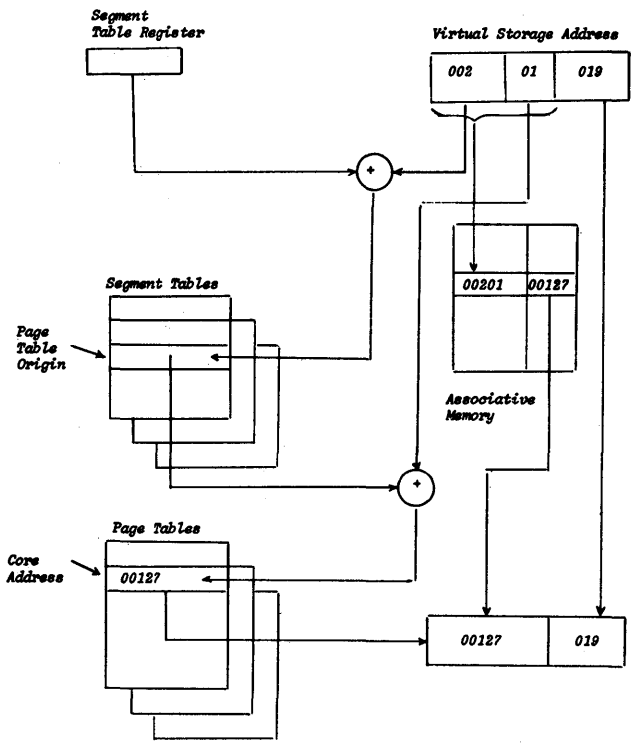


Figure 8—Associative memory

main storage, an entry is made to correspond to its relative location in the user's virtual storage.

The methods vary, obviously, in complexity. An eventual choice on which technique to employ depends solely on the sophistication of the operating system, the access, performance, and utilization required, and the underlying hardware.

Scheduling

In modern operating systems, the supervisor program assumes the highest priority and essentially processes and does the housekeeping for interrupts generated by problem programs and external and I/O devices. In this sense, the supervisor (or the system) is interrupt driven. It is generally hoped that the processing done by the supervisor is kept to a minimum. When the supervisor has completed all of its tasks, it must decide to whom the processor should be allocated. In a single job system, the running program simply retains control of the processor. In a multi-job batch environment, where the system is performance oriented but not response oriented, the processor is usually given to the highest priority job that demands it. This philosophy is generally termed multiprogramming as discussed previously.

In a time-sharing environment, performance is measured in terms of terminal response, and processor scheduling is oriented towards that end. Thus, a user is given a slice of processor time on a periodic basis—frequently enough to give him the operational advantage of having a machine to himself. The scheduling philosophy is influenced by the user environment (i.e., compute-bound jobs, small jobs, response-oriented jobs) and the method of storage management. Three strategies have been used frequently enough to warrant consideration. The most straightforward method is *round robin*. Jobs are ordered in a list on a first-in-first-out basis. Whenever a job reaches the end of a time slice or it can no longer use the processor for some reason, it is placed on the end of the list and the next job in line is given a slice of processor time. A strict round robin strategy favors "compute" jobs and "terminal response" jobs equally and tends to be best suited to a core resident storage management system. With an *exponential scheduling* strategy, several first-in-first-out lists are maintained, each with a given priority. As a job enters the system, it is assigned to a list on the basis of its storage requirements—with lower storage

requirements being assigned a higher priority since they facilitate storage management. The scheduling lists are satisfied on a priority basis, no list is serviced unless higher priority lists have been completed. Terminal (or response) oriented jobs are kept in the highest priority list—thus assuring rapid terminal response. If a job is computing at the end of its time slice, then it is placed at the end of the next lowest priority list. However, lower priority lists are given longer time slices, of the order $2t$, $4t$, $8t$, . . . , so that once in execution, a compute-bound job stays in execution longer. Exponential scheduling has "human factors" appeal in that a terminal-oriented user, who gets frequent time slices, is very aware of his program behavior whereas the program behavior of a compute-bound user is generally transparent to him. One of the biggest problems in processor scheduling is the difficulty in developing an algorithm to satisfy all users. The *schedule table* strategy is an attempt to do that. Each user is given a profile in a schedule table. When a job enters the system, it is assigned default values. As the job develops a history, however, the table values are modified according to the dynamic nature of the program. The scheduler is programmed to use the schedule table in allocating the processor while satisfying both user and installation objectives. The schedule table approach is particularly useful in a paging environment where certain programs require an excess of pages for execution. Once the required pages have been brought into main storage, then the job can be given an appropriate slice of processor time.

Scheduling strategies differ to the extent that a different one probably exists for each installation that is developing one. As such, scheduling algorithms continue to be the object of mathematical description and analysis by simulation.

THE LITERATURE

There are a wealth of good papers on operating systems in the computer literature. In fact, the volume is so great that a literature survey would invariably do injustice to a great many competent authors. In spite of this initial disadvantage, a sample of interesting papers will be mentioned.

Dynamic storage allocation, storage hierarchy, and large capacity storage have been studied in detail by Randell and Kuehner[14], Freeman[15], Lauer[16], and Fikes, Lauer, and Vareha[17].

Performance, program behavior, and the analysis of system characteristics have been reported by Belady[18], Fine, Jackson, and McIsaac[19], Wallace and Mason[20], Coffman and Varian[21], Randell[22], Dennis[23], Denning[24,25,26], Stimler[27], Madnick[28], Habermann[29], Estrin and Kleinrock[30], Shulman[31], and Belady, Nelson, and Shedler[32].

In addition to those already referenced, many operating systems have been implemented for experimental and productive purposes. Still other papers give a survey of multiprogramming and time-sharing techniques. Representative literature in these areas is: Wegner[33,34], O'Neill[35], Arden, Galler, O'Brien, and Westewelt[36], Badger, Johnson, and Philips[37], Schwartz, Coffman, and Weissman[38], Mendelson and England[39], and Kinslow[40].

Eventually, all use of an operating system reduces to a problem in man-machine communication and two important papers by McKeeman[41] and Perlis[42] should be listed in a survey such as this.

Lastly, two compendiums of papers on time-sharing systems have been published by General Electric and IBM. The GE collection entitled, *A New Remote-Accessed Man-Machine System*, describes the MULTICS system at M.I.T. and contains the following papers:

> Corbato, F. J., and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System."
>
> Glaser, E. L., J. F. Conleur, and G. A. Oliver, "System Design of a Computer for Time Sharing Applications."
>
> Vyssotsky, V. A., F. J. Corbato, and R. M. Graham, "Structure of the MULTICS Supervisor."
>
> Daley, R. C., and P. G. Neumann, "A General-Purpose File System for Secondary Storage."
>
> Ossanna, J. F., L. E. Mikus, and S. D. Dunten, "Communications and Input/Output Switching in a Multiplex Computing System."
>
> David, E. E., and R. M. Fano, "Some Thoughts about the Social Implications of Accessible Computing."

The IBM collection entitled, *TSS/360 Compendium*, contains the following papers and reports:

> Lett, A. S., and W. L. Konigsford, "TSS/360: A Time Shared Operating System."
>
> Martinson, J. R., "Utilization of Virtual Memory in Time Sharing System/360."

McKeehan, J. B., "An Analysis of the TSS/360 Command System II."

Johnson, O. W., and J. R. Martinson, "Virtual Memory in Time Sharing System/360."

Lett, A. S., "The Approach to Data Management in Time Sharing System/360."

SUMMARY

Seven properties were introduced for the description, classification, and comparison of operating systems: access, utilization, performance, scheduling, storage management, sharing, and configuration management. On the basis of these properties, the following types of operating system were identified: multiprogramming, hypervisor multiprogramming, time sharing, virtual systems, and tri-level operating systems. Lastly, two major areas of resource management were discussed: storage management and scheduling. Generally, storage management techniques can be classified as to whether they apply to multiprogramming or time-sharing—although the dividing line is not well-defined. Multiprogramming techniques presented were: fixed partition, region allocation, and roll in/roll out. Time-sharing techniques included: core resident, swapping, and paging. Scheduling methods are similarly related to either multiprogramming or time sharing. After a brief discussion, the following time-sharing scheduling philosophies were introduced: round robin, exponential, and the schedule table.

Although formal methods have not been applied to any great extent to operating systems, the interest level is high and many related papers exist in the literature. Operating system technology continues as one of the more challenging areas in the field of computer science.

REFERENCES

1 H KATZAN
Advanced programming: Programming and operating systems
Van Nostrand Reinhold Company 1970

2 A J CRITCHLOW
Generalized multiprogramming and multiprogramming systems
Proceedings of the Fall Joint Computer Conference 1963

3 S ROSEN
IBM operating system/360 concepts and facilities
Programming Systems and Languages McGraw-Hill Book Company 1967

4 T M DUNN J H MORRISEY
Remote computing—An experimental system. Part I: External specifications
Proceedings of the Spring Joint Computer Conference 1964

5 A D FALKOFF K E IVERSON
APL/360 user's manual
IBM Thomas J. Watson Research Center Yorktown Heights New York 1968

6 J G KEMENY T E KURTZ
Basic
Dartmouth College Computation Center Hanover New Hampshire 1965

7 F J CORRATO V A VYSSOTSKY
Introduction and overview of the MULTICS system
Proceedings of the Fall Joint Computer Conference 1965

8 W T COMFORT
A computing system design for user service
Proceedings of the Fall Joint Computer Conference 1965

9 C T GIBSON
Time-sharing in the IBM system/360: Model 67
Proceedings of the Spring Joint Computer Conference 1966

10 A S LETT W L KONIGSFORD
TSS/360: A time-shared operating system
Proceedings of the Fall Joint Computer Conference 1968

11 N WEIZER G OPPENHEIMER
Virtual memory management in a paging environment
Proceedings of the Spring Joint Computer Conference 1969

12 *An introduction to CP–67/CMS*
IBM Cambridge Scientific Center Report 320–2032 Cambridge Massachusetts 1969

13 F J CORBATO et al
The compatible time-sharing system
The MIT Press Cambridge Massachusetts 1963

14 B RANDELL C J KUEHNER
Dynamic storage allocation systems
Communications of the ACM May 1968

15 D N FREEMAN
A storage-hierarchy system for batch processing
Proceedings of the Spring Joint Computer Conference 1968

16 H C LAUER
Bulk core in a 360/67 time-sharing system
Proceedings of the Fall Joint Computer Conference 1967

17 R E FIKES H C LAUER A L VAREHA
Steps toward a general-purpose time-sharing system using large capacity core storage and TSS/360
Proceedings of the 1968 ACM National Conference

18 L A BELADY
A study of replacement algorithms for a virtual storage computer
IBM Systems Journal Volume 4 No 2 1966

19 G H FINE C W JACKSON P V MC ISAAC
Dynamic program behavior under paging
Proceedings of the 1966 ACM National Conference

20 W L WALLACE D L MASON
Degree of multiprogramming in page-on-demand systems
Communications of the ACM June 1968

21 E G COFFMAN L C VARIAN
Further experimental data on the behavior of programs in a paging environment
Communications of the ACM July 1968

22 B RANDELL
A note on storage fragmentation and program segmentation
Communications of the ACM July 1969

23 J B DENNIS
Segmentation and the design of multiprogrammed computer systems
Journal of the ACM Volume 12 No 4 1965

24 P J DENNING
The working set model for program behavior
Communications of the ACM May 1968

25 P J DENNING
A statistical model for console behavior in multiuser computers
Communications of the ACM September 1968
26 P J DENNING
Thrashing: Its causes and prevention
Proceedings of the Fall Joint Computer Conference 1968
27 S STIMLER
Some criteria for time-sharing system performance
Communications of the ACM January 1969
28 S MADNICK
Multi-processor software lockout
Proceedings of the 1968 ACM National Conference
29 A N HABERMANN
Prevention of system deadlocks
Communications of the ACM July 1969
30 G ESTRIN L KLEINROCK
Measures, models, and measurements for time-shared computer utilities
Proceedings of the 1967 ACM National Conference
31 F D SHULMAN
Hardware measurement device for IBM system/360 time-sharing evaluation
Proceedings of the 1967 ACM National Conference
32 L A BELADY R A NELSON G S SHEDLER
An anomaly in space-time characteristics of certain programs running in a paging machine
Communications of the ACM June 1969
33 P WEGNER
Machine organization for multiprogramming
Proceedings of the 1967 ACM National conference
34 P WEGNER
Programming languages, information structures, and machine organization
McGraw-Hill Book Company 1968
35 R W O'NEILL
Experience using a time-sharing multiprogramming system with dynamic address relocation hardware
Proceedings of the Spring Joint Computer Conference 1967
36 B W ARDEN R A GALLER R C O'BRIEN
F N WESTERVELT
Program and addressing structure in a time-sharing environment
Journal of the ACM Volume 13 No 1 1966
37 G F BADGER E A JOHNSON R W PHILIPS
The Pitt time-sharing system for the IBM systems 360
Proceedings of the Fall Joint Computer Conference 1968
38 J I SCHWARTZ E G COFFMAN C WEISSMAN
A general purpose time-sharing system
Proceedings of the Spring Joint Computer Conference 1964
39 M J MENDELSON A W ENGLAND
The SDS SIGMA 7: A real-time, time-sharing computer
Proceedings of the Fall Joint Computer Conference 1966
40 H A KINSLOW
The time-sharing monitor system
Proceedings of the Fall Joint Computer Conference 1964
41 W M MCKEEMAN
Language directed computer design
Proceedings of the Fall Joint Computer Conference 1967
42 A J PERLIS
The synthesis of algorithmic systems
Proceedings of the 1966 ACM National Conference

Computer resource accounting in a time sharing environment

by LEE L. SELWYN*

Boston University
Boston, Massachusetts

INTRODUCTION

The past several years have witnessed major stages in the evolution of time sharing service suppliers toward the (perhaps) ultimate establishment of a computer utility or utilities that will, presumably, resemble other public utilities in many ways. This paper is concerned with the development of managerial accounting techniques that will enable suppliers to broaden their range of services and allow some of them to evolve into vertically integrated information service organizations.

Background

The early time sharing suppliers provided a relatively narrow range of services; the general-purpose systems usually provided access to but a small number of languages and virtually no proprietary software. Indeed, if the latter was available, access to it was provided to customers of the service at no additional charge. Other services provided *only* access to some proprietary applications package, and usually did not offer the generality of access to a programmable service. However, with respect to this latter type of supplier, a charge was indeed imposed for access to the applications software, although it was embedded in the overall cost of the service.

The particular pricing policy established by any one supplier was, perhaps as often as not, forced upon it by some limitation in the computer resource accounting mechanism associated with the time sharing operating system. Hence, many firms "lived with" some schedule of charges that were almost certainly suboptimal. This, in many instances, resulted in substantial limitations upon the overall variety and type of computing services they could provide.

In an earlier paper,[1] the author, along with D. S. Diamond, considered the implications of various types of services upon the pricing policies of a time sharing service supplier. Several possible strategies were considered at that time. These included:

- Transaction-based charges for access to some applications package or data base, where the approximate quantity of computing power required for a given transaction was either (a) reasonably predictable, (b) an insignificant part of the total costs associated with provision of the service, or (c) negligible with respect to the "value" of the service to its end-user.
- Prices based upon resource usage (e.g., cpu time, connect time, core residence, etc.) for (a) general-purpose programmable services, (b) for access to a proprietary applications package where the quantity of computing resources consumed is unpredictable, highly variable, and a major component of costs. (Of course, the resource prices for access via the proprietary package could be higher than for general access to the system.)
- Flat rate for unlimited access to the system, or perhaps a variation on flat rate, such as elapsed connect time. This may be appropriate in cases where the nature of use of the system was sufficiently limited such that this type of rate structure would not result in abuse, or in cases where the charge for a unit of service was so small that the cost of accounting for service was prohibitive. (It should be noted that, with the exception of a poorly designed operating system, such conditions are rather difficult to imagine in practice.)

Whatever pricing structure is eventually adopted

* The author was a research participant at Project MAC, Massachusetts Institute of Technology, which provided partial support for much of the work reported here. He is presently Assistant Professor, College of Business Administration, Boston University.

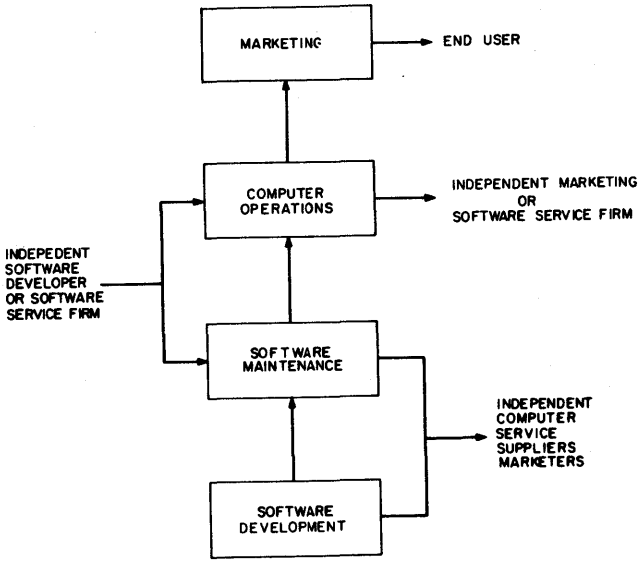


Figure 1—Organization of an information service firm

must be the result of consideration along a number of dimensions. The pricing structure must be "market oriented" so as to satisfy the needs and objectives of customers. The user must be made to feel as if he is paying only for his share of the computer, that activities of other users do not affect his charges. He wants to be able to predict, with some degree of accuracy, what his costs will be. He will base his purchase decision, presumably, upon the value of the proposed computing service to his organization. A second dimension is "operations oriented." The pricing structure must induce users to behave in a manner that is consistent with the best interests of the operators of the facility. If it is found desirable to have a relatively level load on the system, then lower prices must prevail in the less desirable times of the day or week. Users must not be permitted to "hog" the machine, even if they have the funds to pay for the service, since this could cost the computing service other customers who balk at the (perhaps temporary) degradation of service they receive.

Thus, a pricing structure for a computer utility must be flexible enough to handle a variety of service types, must be accurate to a point that satisfies customers' needs for fairness and predictability, must encourage the use of proprietary software with royalties or other payments accruing to the owner, and must be consistent with the requirements of a well oiled operating procedure. The present paper considers the requirements of a managerial accounting and control mechanism to support such a pricing structure, with the stated requirements.

The next section considers the management information requirements of time sharing firms, in light of the current evolution of the industry. Specific design objectives of such an accounting mechanism are then examined in the following section. Finally, the paper concludes with discussions of implementations of such systems by the author—one on the IBM 7094 Compatible Time Sharing System at Project MAC, MIT, and the other on the PDP–10 system operated by Codon Computer Utilities, Inc.

EVOLVING NATURE OF THE TIME SHARING BUSINESS

Services offered by time sharing firms

Where the early time sharing service suppliers provided only a limited range of services, it is becoming increasingly clear that the suppliers of the future will offer a much wider range of services at a much broader level. In effect, a time sharing firm may be thought of as a vertically integrated information service organization with activities ranging from the production of the raw computing power, the development of applications programs and other software, the maintenance of such software, and the retail marketing of its product to the end-user. Figure 1 illustrates a possible organization of an information service firm that provides all of these types of services.

One of the more significant developments in the time sharing industry has been the entry of firms that specialize in some subset of these four major activities. Thus, a software developer may only write an applications package, and may then turn over its maintenance, marketing and operation to an information service organization. Alternatively, the same software developer may perform his own maintenance and marketing, and use the time sharing service only as a source of computing power. The time sharing service, on the other hand, may choose to establish its own retail marketing outlets, or may instead sell its services to a retailer who will assume the marketing risks and rewards within, perhaps, a particular geographical region.

As the industry continues to develop along these lines, the nature of particular arrangements made between its members will grow increasingly more complex. The software developer that chooses to do his own maintenance and marketing will perhaps pay the time sharing supplier for the computing resources he has consumed and then go on to charge his own

customers at whatever rate is appropriate. Alternatively, he may license a package to a time sharing house either on a flat rate or a royalty basis; in either case the two firms must somehow keep track of the use of the subsystem by the ultimate customer.

The necessary record keeping associated with these various levels of activity could, of course, be done at each level; the computer operations area (or firm) could simply measure the quantity of computing resources consumed by each of its customers, which could be end users but might also be software owners' packages, independent marketing departments or firms, etc., and render statements accordingly. The software owner would then develop his own accounting system to measure the use of his subsystem by each of his customers; the marketing organization would similarly have to account for resources used by each of its end-users, and so on. As an alternative, a single information system may be constructed that provides for appropriate managerial and financial control at all levels.

The author began work on the development of such a system while at Project MAC at the Massachusetts Institute of Technology, and has since designed a more complete information system structure for the DEC PDP–10 while serving as a consultant to Codon Computer Utilities, Inc., of Waltham, Massachusetts. The principal design objectives, features and capabilities of these systems are described in the following section.

DESIGN OBJECTIVES

We have already suggested a rationale for the development of an information system for time sharing services that takes account of the vertically integrated nature of the business. Such a system must be based upon several key design objectives, which will be discussed presently.

Computer resource allocation

The system must provide a mechanism whereby it is possible to allocate access to computer resources among the various end-users, subsystems, retailers, and in-house software development efforts.

Even the largest time sharing computers in operation today can support but a mere handful of simultaneous users; in most cases under 50 and in virtually no instance over 100. As a result, the actions of any one user on the system can, and often do, have a noticeable impact upon all of the other users in the community. In an economic sense, the users are "oligopsonists," i.e., there are relatively few buyers (of time sharing service) in the (captive) market associated with the single "monopolistic" machine. Of course, this description of the market structure does not hold for even the medium run, let alone the long run. There are now numerous time sharing service suppliers, and numerous customers, such that something more closely resembling a competitive market exists. However, in the very short run, a given user is, in effect, captured by a given system. As an oligopsonist, his actions affect both the system and the other users.

Hence, his access to the system must somehow be controlled, irrespective of his ability to pay. As an example, a user desirous of obtaining the entire machine for a period of, perhaps even five minutes would create much unrest among the other users. As a result, the time sharing monitor will normally use some sort of round-robin or related method of scheduling jobs so as to prevent a user from obtaining this much service in this short a time. But what of the user who requires some large number of simultaneous lines and many connect hours, plus perhaps some very large quantity of mass storage, over a relatively short period of time, perhaps a week or two. This could place an unnecessarily heavy load on the system and once again cause some unrest among the other users. This would, of course, be perfectly reasonable, from the point of view of system management, if this new heavy demand were permanent; but if it is only a very temporary thing, then the system, and the remaining customers, must be protected. In effect, some rationing scheme is indicated. By such a mechanism, a user reserves, in advance, that quantity of system resources he is likely to require during some time interval, perhaps a month. By this technique, system management may allocate available resources among its customers in an attempt to even the load on the system. For example, the day may be divided into several periods, such as peak and off-peak, and different rations might apply to each such period for any customer. By this technique, the user can be assured of more even levels of service at all times, and system management may more accurately forecast its facilities requirements.

Flexibility in pricing

Although the resource management system will use the firm's pricing structure as a basis for its record-keeping, it should by no means be bound or limited by the specific pricing mechanism that has been selected by management. There are several reasons for this. Perhaps the most obvious is that the marketing strategy of the firm, and hence its pricing policy, may be subject to some modification as time goes by. Moreover, if in

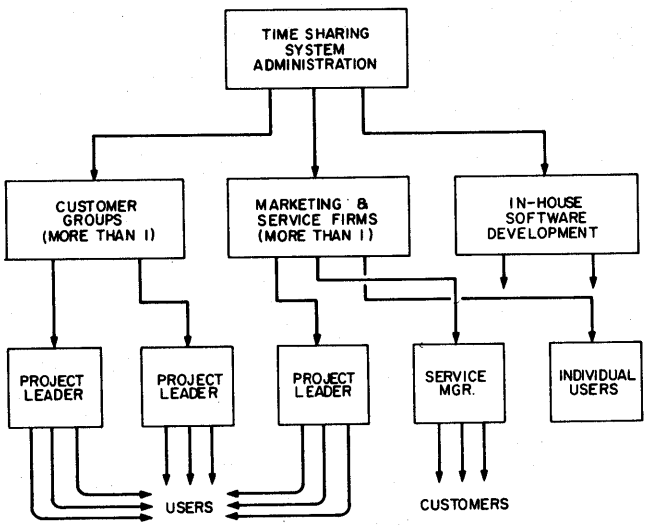


Figure 2—Resource management structure

fact the firm is going to deal with independent software developers, independent retailers, or both, it will not want to impose its pricing structure upon these firms in the latters' dealings with their own customers. Hence, the system should be capable of supporting a variety of pricing policies at the same time. It should have the capability of charging the intermediary at prices established by mutual negotiation or any other means, and then permit the intermediary to impose virtually any sort of charge upon the latters' customers. This separation of wholesale and retail types of charges should be reflected in all other parts of the system, from resource allocation to billing.

Decentralized management

Since the time sharing supplier will be dealing with some intermediaries, it is necessary that the latter be provided with some resource management and administration tools, thereby enabling control over the activities of the intermediary's customer. Moreover, the actual customers of both the time sharing firm and its intermediaries can greatly benefit from such local resource management facilities. Thus, a customer's project leader should be able to directly manage use of the computer among the several participants in his project. He should be permitted to allocate some set of resources given to him among users in his project. Similarly, an intermediary should be allocated some pool of resources and should be permitted to directly allocate these among its customers according to whatever method it wishes.

In effect, the time sharing supplier need deal only with intermediaries, its own marketing organization, and customers. Each may then manage its own use of the computer and allocate the ration of system resources among those users and activities subject to its control. Such a resource management structure is illustrated in Figure 2.

Access to system resources and services

The foregoing implies a fairly tight method of controlling access to the computer. This is especially true where price differences apply to different classes and types of services. Thus, a user of a proprietary subsystem who pays a higher price for system resources than does a general time sharing service user must be restricted to use the system only through the account specifically established for this purpose. Similarly, a general purpose time sharing service user must be prevented from gaining access to any proprietary service for which some unusual charge applies, without first establishing an account for the use of such services. For any end-user, the operating system must know (1) who is responsible for charges (i.e., a customer or an intermediary), (2) what type of service this user has subscribed for, (3) what resources have been allocated to this end-user, and (4) which price shcedule applies to the user and which to the responsible account (which may, of course, be the same).

Detail of system usage

It is possible for a time sharing system to keep a detailed log of all transactions that take place within it. It is, for example, possible for every system command to create a message, stored somewhere in the system, that indicates the time, date, name of the command, the identity of the user, and the quantity of resources used in the execution of the command. This would, however, be a fairly costly procedure, in that system overhead would be increased to handle each of these message creation and storage. From a managerial point of view this may be quite unnecessary. However, there are certain types of individual services which must be accounted for in great detail, both from the standpoint of billing and collection of revenues, and from the standpoint of operations analysis and control. Thus, services provided by proprietary software that is to be charged on the basis of transactions performed rather than computer resources consumed must be maintained in great detail so that the customer may know how he has spent his money and that the software developer may examine the relative efficiency

of his programs. Moreover, the time sharing system management will want this information to compute that portion of the subsystem's total revenues to retain for providing service.

Subsystem usage accounting

Proprietary subsystems that offer services to be charged on a "value of service" or transaction basis necessarily have pricing structures radically different from any resource-based rate schedule. In fact, just about the only similarity between any two such subsystems is that they both compute their rates according to some unique set of rules determined by the developer of the application package. The resource accounting mechanism must not only cope with such a variety of structures, but should additionally encourage innovation on the part of software developers to use any arbitrary mechanism that they desire. Hence, the accounting system should permit the subsystem to compute its charges and to report to the accounting mechanism these charges together with some identification of the nature of the service provided to the end-user. The accounting system should be able to record these subsystem-imposed charges in the end-user's account, retain the details of the transactions for billing and analysis purposes, and to impose upon the subsystem itself a charge based upon the resources consumed in completing the user-initiated transaction. In effect, the subsystem becomes the customer of the time sharing system, the end user is charged by the time sharing system on behalf of the subsystem. (That is, the time sharing firm offers, as a service to software developers, a means by which it will handle all of the paper work associated with the software developer's customers' accounts.)

IMPLEMENTATION

Many of the features mentioned here were included in a resource accounting system implemented by the author on the IBM 7094 Compatible Time Sharing System developed at Project MAC at the Massachusetts Institute of Technology.[2,3] The "BUYTIM" resource allocation system was designed to operate wholly within the CTSS operating system, and required no monitor-level modifications to the CTSS software. The CTSS implementation is described below.

When Project MAC started charging for CTSS usage in January of 1968, a need arose for some changes in the mechanism by which users requested additions to or changes in their set of allotments of computer resources. Formerly, these resources were allocated directly—that is, by specific allocations to each user, of time, divided into five shifts, and disk records for storage of user files. Under the new scheme, users receive dollar budget allocations, from some sponsored project, and use these funds to purchase resources on the CTSS System. The mechanism described here serves to give the individual user more direct control over the means by which his available funds are spent.

The system provides for computer resource management at a number of levels, ranging from top-level administration down to the level of the individual user. Under the pre-existing resource management system this function was assumed, within the MIT Information Processing Center, at two levels: Administration and User Group Leader. The Administration held ultimate responsibility for computer resource allocation; it classified all users into one of about twenty user groups, and determined the particular mix of resources that were to be made available to each of the groups. The Group Leader was responsible for apportioning the resources allotted to his group among its members. This responsibility is not altered under the BUYTIM set-up. However, what is provided to the Group Leader is a means for further delegating his responsibility—and power—to individual members of his group at two principal levels: the Problem Number Leader and the Individual User.

The Problem Number Leader is afforded the capability of apportioning resources allotted, by the Group Leader, to his problem number among its members, a relationship vis-a-vis the Group Leader not unlike the latter's relationship to CTSS Administration. The Problem Number Leader is further afforded the capability of delegating some of his authority to individual members of the problem group, with several levels of decision-making capability. That is, the problem number leader may designate another user in the problem group to have identical capabilities as himself, or he may choose to limit these capabilities in some way. At the limit, the individual may be given the ability to make changes in only his own account, or may be denied even this capability.

User preferences for computer resource allocations are subject to two types of limitations: pricing and rationing. Resource prices have been established by CTSS administration and govern the allocation of time and disk, as well as of several other "commodities." Only the time and disk are covered by this system. In addition to the constraints of the costs of CTSS resources, users are further constrained by several restrictions which limit their ability to spend their allocation of funds as they might please. First, each user group is given a set of Group Limits containing the maximum amounts of each commodity that may be

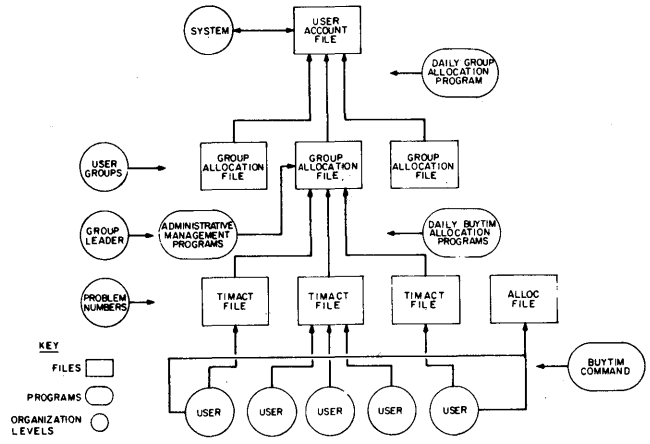


Figure 3—BUYTIM system files and programs

allocated among the group's members. Under the BUYTIM system, the group leader may further break this set of limits down among individual problem numbers in the group, or may have all problem numbers share the common pool of resources available to the group, or a combination of these techniques. To further restrict the user's freedom, the group leader may establish a maximum increment by which a user may increase his allocation over his actual usage to date. At the beginning of each calendar month individual user *time* allocations are reduced to nominal beginning of month levels, also established by the group leader for each user, thus requiring the latter to repurchase additional requirements for the new month.

The BUYTIM system communicates with the CTSS Administrative management programs by updating the Group Allocation File (GAF), a method used by all of the individual user groups within CTSS. As a result, it is not necessary for all user groups to subscribe to and implement the BUYTIM system; its use is totally transparent to the pre-existing allocation mechanism.

Two new file types were created for the purposes of effecting communication between the individual user and the GAF, and a number of programs were written to permit suitably formatted and protected modifications of these files. The principal file in the system is the 'TIMACT' file. There is one such file for each problem number in a subscribing user group. In addition, there is a file of the form "GRPXX ALLOC" for each user group. All of these files are maintained in a special directory in Private, Protected mode. Hence, these files are accessible directly to the Group Leader, and are made accessible to users only through a special privileged CTSS command which may modify these files. Figure 3 illustrates the file structure in this system.

The use of these TIMACT files makes it possible for the Group Leader to subdivide the overall Group allocations of time and disk into blocks available to each problem number within the group. However, in some cases, particularly where the problem number contains only one or a few programmers, this feature may be undesirable, since the overall group limits would be segmented to a point where flexibility to meet individual user needs, within the pricing mechanism, would be seriously restricted. In such cases, the Group Leader may assign a particular problem number to a common pool of resources. This means that, instead of getting problem number limits from the TIMACT file, the limits come from the file GRPXX ALLOC.

The BUYTIM command provides an on-line mechanism for requesting changes in CTSS time and disk allocations. It permits a user to "spend" the funds allotted to him, at the prevailing prices for the several commodities as contained in the file SYSTEM PRICES, subject to availability, and also enables the user to change his password. In addition, BUYTIM provides varying capabilities for project leaders to make changes in funds and computer resource allocations of other users within the same problem number group. A wide range of access privileges are available for this purpose.

Although the individual CTSS user and his project leader are provided with the ability to trade-off among the various CTSS resources at the established prices, this capability is limited to the extent that such resources are available to the project group.

Classes of use of BUYTIM

There are seven distinct classes of use of BUYTIM, each of which affords the classified user with certain privileges, in terms of what types of changes he may request via the command. The class codes for each user are contained in the TIMACT file, which also contains the other account information about the user. The class designations are as follows:

0. No access to BUYTIM whatsoever.
1. User may change his own time and disk.
2. User may change his own time, disk, and password.
3. User may change his own time, disk and password, and may change time and disk of other users in the problem number group.
4. User may change his own time, disk and password, and the time, disk and passwords of other users in the same problem number group.
5. User may change his own time, disk, and password, and the time, disk, and funds of other users in the same problem number group.
6. User may change time, disk, password and funds

allocations of himself and all other users in the same problem number group.

Change codes

There are nine change codes available, plus a termination code, as follows:

Code	*Description*
*t*1	Shift 1 time
*t*2	Shift 2 time
*t*3	Shift 3 time
*t*4	Shift 4 time
*t*5	Shift 5 time
disk	Disk records
pass	Password changes
funds	change in funds allocation
prpass	print user password (class 4 and 6 only)

The '*' is used as a termination code, to denote the completion of a series of changes.

To change time, disk or funds allocations, a user types the appropriate change code followed by the amount of the change in integral minutes of time, disk records, or dollars of funds allocation. The amount of the change may be a signed or unsigned number. If signed (e.g., +25, −40) the present level will be changed by that amount. If unsigned, the number is assumed to be the new level of the allocation, and will thus replace the old one. For example:

TYPE CHANGES: *t*1 +10 *t*3 −5 *t*4 20
disk +50 funds +100 *

This will increase shift 1 time by 10 minutes, reduce shift 3 by 5 minutes, set the shift 4 allocation to 20 minutes, increase the disk allocation by 50 records and increase the funds allotment by $100.

Charges are levied by BUYTIM on the basis of allocation, rather than usage. An unused allocation may be returned for full credit at the prevailing PRICES at the time of the return. In the case of time, the charge imposed (or credited) is 1/60 of the prevailing hourly rate for each integral minute of time allocation purchased (or returned). In the case of disk space, the charge (or credit) is based upon 1/30 of the prevailing rate per disk record per month times the number of days left in the month. Thus, a disk allotment is paid through the end of the month, and credits are figured from the date the space is released until the end of the month. BUYTIM does not consider the other charges, such as those for the monthly account maintenance fee and the U.F.D. charge. These should be estimated by the Group Leader and subtracted from the funds balance appearing in TIMACT.

Because BUYTIM charges on the basis of allocation rather than usage, the dollar balances obtained from these alternate methods will usually differ somewhat. However, note that BUYTIM is provided for the convenience of users and Group Leaders, and where a discrepancy exists the figures based upon usage, as provided by CTSS charge statements, will prevail.

BUYTIM will reject several kinds of transactions:

Unauthorized use

1. Unauthorized use of BUYTIM (class 0 user).
2. Attempt to change account of another user (classes 1 and 2).
3. Attempt to change specific items of another user or of own account not permitted by class designation.

Allocation restrictions

4. Attempt to increase allocation of time or disk above maximum increment set by Group Leader or Problem Leader.
5. Attempt to reduce allocation below current usage.
6. Insufficient funds.
7. Increase in allocation exceeds available resources.

There are several capabilities available to the Group Leader that are not available to the individual user or problem number leader within the BUYTIM command. Several other programs were written to facilitate these functions by the Group Leader. For example, the group leader may add or delete users, may assign various types of access privileges and restrictions, and may apportion the group resource limits among the individual problem numbers in the group.

All changes made by users, problem leaders, and group leaders are recorded in the appropriate TIMACT file (although the group leader may occasionally modify the Group Allocation File directly). The modified TIMACT file records must be posted to the Group Allocation File in order for the CTSS resource allocation programs to recognize the changes. This is accomplished by means of a self-rescheduling Foreground Initiated Background job run each evening by the group leader. (A mechanism in CTSS permits such jobs to be scheduled and run automatically, without the presence of a user at the time the job is run.) Hence, changes made during any given day cannot be recognized by the time sharing system until they have, first, been posted to the Group Allocation File, and second, been copied from the Group Allocation File into the primary system accounting files. This latter activity usually occurs at midnight, also via a self-rescheduling job. Thus, changes made via the BUYTIM mechanism will usually take effect at or around midnight following the change request.

PDP–10 implementation

A major extension of the concepts developed in the BUYTIM system has been designed for the PDP–10 time sharing system in operation at Codon Computer Utilities, Inc.[4] The new features of this extension are described here. It should be noted, however, that unlike the CTSS version, several monitor-level system modifications were required for the PDP–10 design. As a result, the system is not transparent to the operating system, but forms an integral part of it. Besides extending his own concepts, the author wishes to acknowledge the work by Thomas H. Van Vleck of MIT, who developed the overall CTSS accounting mechanism (within which BUYTIM operated) for a number of the ideas incorporated into the PDP–10 version.

Unlike the CTSS case, the newer version was built directly into the time sharing monitor, replacing the manufacturer-supplied resource accounting mechanism, rather than simply operating within it. As a result, any alterations to the user accounts take effect immediately, rather than at some later time when the changes might be posted to the user accounts, as in the CTSS case. Several important new features were, additionally, added.

Dynamic pricing

Under the CTSS version, the principal control on usage, during real-time operation of the time sharing system, was the central processor time consumed by an individual user. Thus, a user might have received an allocation of, say, 20 minutes of prime shift processor time for the current month. When that allocation is exhausted, the user is automatically logged off the computer by the monitor, and is prevented from logging back on during that shift for the duration of the month, or until he can secure from the group leader, or through the use of BUYTIM, additional allocation for the shift. In a commercial environment, the operator of a computer utility may desire to control other resources besides the amount of straight processor time employed by his customers. Moreover, his pricing mechanism may be non-linear, in that lower unit prices may apply for larger quantities of a given resource consumed. Further, depending upon the nature of the customer, (e.g., an end user or an intermediary) the nature of his application, and any special terms negotiated with him in contracting for service, it is conceivable that several different rate schedules may have to be devised and used simultaneously during the real-time operation of the computer. In the PDP–10 version, four distinct types of charges may apply to a user account during a console session.

Central Processor, or "Computation"
Transaction service usage
Connect time
I/O device usage

The computation charge is based upon the processor time and the core residence during execution of the user's job. The applicable rate may be non-linear, in that as core residence increases, the unit charge for a space-time unit (kilo-core-second) may vary.

Certain services of a computer utility may be marketed in terms of the "service" they provide, rather than the quantity of system resources they consume. Such "transaction" services are charged at varying rates, the exact charge being determined by the particular proprietary program that provides the service. The charge may be based upon what is being done, how much of it is being done, and, perhaps in some cases, who is doing it.

Connect time is the elapsed time between login and logout. It may vary according to the type of terminal (e.g., use of the system from a high-speed CRT display terminal may be charged at a higher connect-time rate), and perhaps at a different rate depending upon whether or not the job is in an "attached" state or in a "detached" state, wherein no console is physically connected to the computer for the detached job.

Use of I/O devices, such as line printers, magnetic tape drives, card readers and punches, etc., are charged for at rates applicable to each device. Further, a set-up or minimum charge may also apply in some instances.

In each of the four types of charges, the specific structure may vary among classes of users, as well as the time-shift in which the service is provided.

Under the dynamic pricing technique, each user is given money allocations for each applicable time shift, and is free to spend his allocated limits on any of the four types of services just described. Whenever some service is provided, except in the case of transaction services, the system computes the quantity consumed, determines the applicable rate structure for the customer in the current time shift, and proceeds to charge the account the money cost of the service. If this charge brings the balance for the shift below zero, the user may be logged off the computer, with an appropriate message explaining the reason for this action. In the case of transaction services, the transaction program itself computes the applicable charge and, by a suitable monitor call, conveys this information to the monitor. Specific resource usage (computation, connect time, and I/O device usage) is charged by the monitor to the transaction service, in a special account maintained

for this purpose, and not to the user. No negative balance check is made against the transaction service account. When the transaction service informs the monitor of a charge to the user, the user's account is charged and a negative balance check is made. Control is returned to the transaction service in any event but, where a negative balance condition exists, the transaction program is so informed and is expected to take action. Thus, there is a very important assumption made about the nature of a transaction program. It is considered to be a well debugged program that is fully responsible for all accounting interactions with the monitor on behalf of the user. It must compute the charge, inform the monitor of its conclusion, and take appropriate action in the event the user's funds have been exhausted.

Under the PDP–10 time sharing system, it is possible for the same user to be logged on the system several times simultaneously. Thus, it becomes necessary to coordinate the charges incurred by each of the several possible jobs in simultaneous operation in a single funds balance. Thus, as soon as any one of the jobs logged in under the same project-programmer number does something that results in a negative balance, *all* of the jobs subject to this balance will be logged off the system.

Bills are rendered to a responsible account, rather than directly to the user. Of course, these may be the same person. However, by this mechanism, an independent retailer may render his own bills to his customers. Alternatively, the billing system may prepare such bills for the retailer. The flexible price structure mechanism enables the individual user to be charged, during real-time use of the computer, under a rate structure that may differ from the one for which services will be billed to the responsible account. Thus, an independent retailer may convey to the computer utility his own rate structure, based upon which the utility will charge and prepare bills for the retailer's customers. The wholesale prices charged by the utility to the retailer need not be the same, either in level or structure, as the latter's retail prices.

Accounting for services used

The large variety of things that a user may be charged for in a computer utility system requires an accounting mechanism that can collect, maintain, display and summarize the detail of individual user activities. Moreover, in the case of proprietary transaction services, detail as to the nature of individual types of services provided, their quantities and applicable charges, is most desirable from the point-of-view of the customer. Further, system and application subsystem management will want such detail so as to best analyze the relative efficiencies of the various services offered, to perform market demand studies, and other management analyses.

The accounting system designed for Codon seeks to provide these features. Each user account record maintains a breakdown of the dollar value of resources used in each of the four principal categories and in each of the applicable time shifts. A user may obtain this information for himself from a logged in console by a suitable monitor command.

Thus, during a console session, the monitor maintains a record of the consumption of the four "temporal" resources and, upon logging off the system, reports these figures back to the user account record. Moreover, a record of the individual console session is created, containing the resource usage data, time and date, and other relevant information, and is maintained for later processing and auditing purposes.

In addition to the temporal resources (temporal in that they are consumed over time) the system accounts for use of "spatial" resources, such as mass storage occupancy. The technique here is quite analogous to the scheme employed within CTSS, as described earlier. Besides accounting for disk storage, the monitor will not permit a file to be opened for writing if the quota of disk blocks has been exceeded. When such a condition occurs, the user must either delete some files to free up some space, or have his allocation of disk space increased.

Specific transaction services are recorded as they are provided, by means of records that contain data on the user's identification, the type and quantity of transaction services provided, and the applicable charge. (E.g., preparation of 40 payroll checks @ 20¢ each, $8.00.) At the same time, the charge to be imposed to the user is also added to the transaction service's account for auditing purposes. At the end of an accounting period (e.g., a month) the transaction service records are sorted by user and summary statements are prepared showing the basis for the aggregate transaction service charge.

Usage of input/output devices is also handled by a similar detailed recording procedure. A record is maintained for each access to a particular device, indicating the duration of access and the applicable charge for the service. These records may also be summarized at the close of an accounting period and presented in a detailed statement to the user. System management may also be provided with a detailed picture of the relative demand for access to the various peripherals on the computer.

Thus, the billing process will generate a summary

bill, which provides aggregate charges for each of the four temporal resources, the principal spatial resource, disk, and any other charges related to the account. Further, a detailed statement of specific transaction services supplied may be produced, as may a similar detailed breakdown of I/O device access. Finally, an historical record of the individual console sessions may be generated by the accounting system.

Applications subsystem owners, whose software is offered on a transaction basis by the computer utility, may be provided with a statement of the resources consumed by their respective programs, as well as the revenues generated by their subsystem's customers. A similar type of detail may be provided in the case of usage of peripheral devices.

Finally, it should be pointed out that the accounting system incorporates all transactions between the customer and the utility, and provides a convenient mechanism for posting miscellaneous charges and credits, such as a charge for consulting service or a credit for a system failure, directly to the user's account record maintained on the system.

System access

The nature of access to the computer, by individual users, may be restricted by the system administrators in several ways. In the simplest case, it may be desired to restrict access to only some of the time shifts during which the machine is in operation. This may be handled quite simply by setting the funds allocations in the restricted shifts to zero. The system will not permit a user to log in during these times, since the balance remaining in his account, for the shift, is zero.

Where a user subscribes to the computer utility for a specific proprietary applications subsystem, it may be desired to restrict access to that subsystem. Moreover, access to the subsystem programs must be restricted to only those users who have subscribed to its service. These objectives may be accomplished by a fairly straightforward procedure. First, applications subsystems are accessed by special system commands which perform a login procedure for the user invoking them. If the user is to be restricted to a subsystem, the name, or some other unique identification, of that subsystem is placed in the account record. The normal system login procedure checks to see that no such restriction exists; if it does, login as a general time sharing user is not permitted. The login procedure in each subsystem must verify the equality of the subsystem restriction with its own requirements. For example, a subsystem might accept several legal subsystem identifications in the user account record, but assign different levels of service privileges to the user based upon the particular code that is present in his account. (Of course, a subsystem might not perform any such check, allowing any subscriber to access it and purchase its services.)

On occasion, it may be desirable to limit the number of simultaneous jobs that may be active for a group of users, e.g., a number of individual users all associated with the same customer of the computer utility. This capability permits the utility to offer guaranteed access for a stated number of individual users to a large customer organization. This concept may be implemented in two ways: a guaranteed *minimum* number of lines, or a guaranteed *total* number of lines. In the former, the customer would be guaranteed that he could always have some specified number of lines; if he were using all of these, he could obtain additional lines on a contention basis with other customers who do not have any guaranteed access. In the latter case, the customer cannot exceed his guaranteed number of lines; additional users will be prevented from logging in until another user in the group has logged out.

We have already considered the case of multiple jobs for the same user account. The system has the capability of placing an upper limit on the number of times a user may be logged in. However, for a user in a guaranteed access group, the use of multiple logins on the same user account will count toward the access guarantee for every time a single user is logged in.

Resource management

The preceding sections have described the control capabilities of the PDP–10 design. In order to administer the system, a mechanism has been provided for communicating with the accounting structure at a number of levels, similar, but with some significant extensions, to the BUYTIM system on CTSS described earlier. The program that enables such communication is implemented as an application subsystem that may be made available to virtually all users of the computer, but subject to several distinct levels of access. The same type of management levels that exist in the CTSS version are available here—utility administration, user group leader, project leader (analogous to problem number leader in CTSS) an individual user. In the CTSS version the group leader could allocate individual sets of resource limits to each problem number, or place some or all in a common pool. A similar capability exists here, except that any number of resource pools may be established within the same user group, instead of only the one available in CTSS. Generally, each level of management has greater direct control ca-

pabilities than were available within the CTSS version. The following table summarizes the principal capabilities of each level of resource management, under the PDP–10 version.

RESOURCE MANAGEMENT CAPABILITIES

Level	*Capabilities*
User	May have no access to subsystem, or May do any of the following, as determined by project leader: Alter own resource limits, password, or funds. Alter any or all of the above for other users in same project.
Project Leader	May do any of the above within his project. May assign any of the above levels of permission to users in his project. Add or delete users in his project. Examine the accounts, including passwords, of users in his project. Designate specific applications subsystem access to any user in his project, subject to restrictions imposed by Group Leader.
Group Leader	May do any of the above. Designate project leaders for projects within his group. Assign and remove individual project numbers to and from resource pools. Change allocation limits for projects and resource pools within the group.
Utility Administration	May alter rate structure applicable to individual user accounts. Alter guaranteed access group assignment for individual users, projects and customer groups. Change overall resource limits for individual customer groups. May add and remove project numbers to or from a customer group. May create and destroy customer groups. May alter the number of lines, as well as the nature of, a guaranteed access group. May enter any extraordinary charge or credit to any individual user account. May set, for each user, the number of times he may be simultaneously logged in. May establish and modify the list of subsystems available to the user group.

Security of the System

Several procedures and mechanisms have been incorporated into the design of the resource management system to provide protection from accidental or deliberate sabotage by users. All accounting files are maintained in a manner such that access to them is only possible by means of one of several system commands. Further, these commands are responsible for restricting access to the accounting files according to the level of authority of the user invoking the command. This permission information is maintained in the user account record. When executing the accounting commands, the user is prevented from returning to monitor level until all files have been closed and access has been terminated. Moreover, should a user accidentally be able to get to monitor level, the system will not permit him to do anything except log off the computer.

Similar restrictions apply in the case of use of an application subsystem. Once access has been gained, control cannot be returned to the time sharing monitor unless and until the program so desires. User-initiated interrupts are intercepted by the monitor and control is passed back to the proprietary subsystem then in execution. The subsystem is, of course, responsible for taking whatever action it considers appropriate. A normal exit from such a subsystem implies a logout. If a user wishes to use both the subsystem and the general purpose time sharing service, he must establish separate accounts for each purpose.

We have attempted to present a description of how a managerial accounting information system for a computer utility can significantly expand the scope of activities of such an organization. The approach to system design has been the result of actual administrative experience with such systems over a period of several years. This experience has shown the importance of such capabilities.

As time sharing service organizations become more complex in their structure and diversified in their activities, the need for a well-structured information management mechanism will no doubt become more critical. This implies that time sharing operating systems will, more and more, have to be designed with the necessities of system administration in mind. Operating

systems deficient in this respect will find it difficult to provide the range of services necessary for survival in an increasingly more competitive industry environment.

REFERENCES

1 D S DIAMOND L L SELWYN
Considerations for computer utility pricing policies
Proceedings of 23rd National Conference Association for Computing Machinery pp 189–200 Brandon/Systems Press Princeton New Jersey 1968

2 P A CRISMAN Ed
The compatible time sharing system: A programmer's guide
Second Edition MIT PRESS Cambridge Massachusetts 1965

3 L L SELWYN
BUYTIM: A system for CTSS resource control
Unpublished memorandum Project MAC MIT Rev July 1969

4 *Codon resource allocation system: User's guide*
Codon Computer Utilities Codon Corporation Waltham Massachusetts 1970

Multiple consoles: A basis for communication growth in large systems

by DENNIS W. ANDREWS and RONALD A. RADICE

International Business Machines Corporation
Kingston, New York

INTRODUCTION

The intent of this paper is to discuss the development of multiple consoles which on the surface appears to be a simple and straightforward concept. This concept, however, should not be considered an ending in itself, it should be viewed as a basis from which a communication network can grow. Justification for more than one console is supported to show that multiple-consoles are necessary components in most if not all large systems. Beyond the basic and obvious requirements of a multiple console environment lie many different philosophies concerning the utilization of consoles, some of which are explored in this paper.

The recently announced Multiple Console Support (MCS) Option of the IBM System/360 Operating System constitutes one significant approach. Within its own environment MCS leaves room for meaningful growth in the communications network.

THE MULTIPLE CONSOLE CONCEPT

As one looks back over the history of computer evolution the evidence of development in system features shows a steady growth. Increased diversification, throughput capabilities and physical size have all contributed to overloading the communication network. The discontinuous flow of system messages and the proportionate increase in the number of messages with the growth pattern in system features, made the task of the operator demanding to the point of physical and mental exhaustion.

An informal study revealed that three messages per minute based on an average message mix is the maximum number an operator can handle without delaying system throughput. Information only messages were the easiest for the operator to handle, because all he had to do was remember what was outputted to him regarding the system status. But as the number of decision and action type messages increased so increased the complexity of the operator's job. His role in the man-computer communication network became more vital as the system and the number of system messages continued to grow. It soon became obvious any breakdown in the response time of the operator would only degrade the power and throughput of the system.

What was a user to do? He wanted more system features and was aware that with this increased power the system would need a corresponding increase in operator-computer communications. As one user described it, "With priority scheduling and with multipartition or multiprocessor operations, the operator at the console is in the position of trying to drink from a firehose." Another user found that their IBM 1052 typewriter console output was approximately 4 million characters a month based on a 24-hour day, 7 days per week operation, or 100 characters per minute. And considering that the maximum rate of a 1052 is 14.8 characters per second and allowing 500 milliseconds for a carrier return this meant that the console was busy on an average of one-eighth of each minute. However, realizing that console output rates follow probability the situation arose too often where the console had all it could do to keep up with message output. At these times, the operator was kept abnormally busy trying to interpret what was going on. Then, if required operator communication requests are considered, such as displaying information about job activity on the IBM System/360 Models 65 and up, chances were that what the operator got back in reply was a history of what existed rather than what was presently happening in the system. The IBM 2250 Display Console helped somewhat in offsetting the time required to display these types of informational messages.

Another problem was that even if the operator, or operators with the larger systems, could keep up with

the flood of output messages they were kept active trying to satisfy peripheral requests made by the system. Add to this fact that with some users it is either necessary or desirable to have peripheral equipment located in other rooms or even other floors, the inconvenience of satisfying peripheral requests is compounded. Voice communication between operators was previously the only way of handling many of these distant peripheral demands made by the system. But voice communication proved faulty either through unintelligible operator enunications or because of the lack of readable copy for the peripheral operator who might forget a request when his area might be overly busy. The result was needless I/O delay.

Another significant point was that all communication was dependent on one relatively inexpensive console unit. If for some reason this device should fail, all communication would stop and the system would eventually stop.

The solution to this overloaded operator and overloaded console problem was to have multiple operators at multiple consoles. Now, for example, all the operator stationed in the tape pool had to do was scan the console output near his station for messages related to the tape pool. Clearly this was a trivial solution, but it was nonetheless a solution. The inherent problem, of course, was that each operator received all system messages. We shall see later how a specific implementation of the multiple console concept can eliminate this problem of message duplication at all active consoles.

DESIRABLE ASPECTS OF A SOLUTION

What was needed to satisfy the growth in system power was an encompassing communications network that would be versatile enough to accommodate future system growth while affording immediate solutions to the present day problems inherent in man-computer interactions.

At this point then, let us enumerate what should be the desirable aspects of such a solution:

(1) Reliability—A system component as vital as the communication network should afford safeguards against failure whenever possible. Also, there should be a guarantee that every generated message gets delivered.
(2) Flexibility—The network should have the capability of molding to fit the immediate needs of the system, and allow room for future growth.
(3) Standardization—To guard against the heretofore haphazard growth in the communication area, formalized procedures should be developed to eliminate needless duplication of messages and guard against multiple formats of messages.
(4) Communication Power—Presently, a major deficit is in speed of communication. The solution should offer a definite increase in output capability. This, to have value, would have to be coupled with a routing capability which could deliver messages to their most appropriate point.
(5) Auditability—The network should be able to maintain a trail of past communications to aid in system analysis and evaluations.
(6) Security—A user desiring to keep his computer interactions private should be given that capability.

MULTIPLE CONSOLE SUPPORT

One answer to the multiple console concept was provided by IBM with the Multiple Console Support (MCS) Option, made available in Release 18 of the IBM System/360 Operating System. Previous attempts at multiple consoles within IBM were made through the ASP–HASP spooling systems, but the MCS Option was the first supported package that not only made multiple consoles a working idea but provided a communications network that could grow with the system. Let us now look at some of the highpoints of this MCS Option.

As its name implies, MCS enables the Operating System to be configured with multiple consoles, with each console performing one or more dedicated functions.

The primary means of *routing messages* to their appropriate point is via 'routing codes' assigned to each message. One or more routing codes may be specified to indicate to which functional area a message is to be sent. More than one routing code may be assigned to a message.

Descriptor codes provide multiple means of message presentation or message deletion from display type devices. These codes provide the individual console device support with the means of determining how a message is to be printed or displayed, and how a message may be deleted from a display device.

The user specifies the *console configuration* when building the system. He may dynamically alter the configuration during system operation, however. One console must be specified as the Master Console, where all commands are valid. All other consoles are specified as secondary consoles with each console having a command entering authority and assigned routing codes.

Secondary Consoles are additional consoles (local or remote) to which selected messages are routed. One

console may handle more than one routing code, and the same routing code may be handled by many consoles. The user specifies which operator commands and routing codes will be authorized for each secondary console when the system is built.

Alternate consoles provide backup capability when the original console device is inoperative. An alternate console can be a secondary console or the Master Console. MCS requires that an alternate console be specified for the Master Console. If an alternate console is not assigned to a secondary console, the Master Console will be assigned as the alternate. This alternate console concept enhances the network reliability by ensuring that messages are not lost when one console goes down or offline. Initially, each console's alternate is assigned during system definition but can be dynamically changed during operation.

Six console types are presently supported by MCS:

(1) IBM 1052 Printer Keyboard Model 7 with a 2150 Console.
(2) IBM 1052 Printer Keyboard Model 7 with a 1052 Adapter.
(3) Composite Reader/Printer or Reader/Punch combinations.
(4) IBM 2250 on IBM System/360 Models 50, 65, 75, and 91 using MVT. (Display Screen)
(5) IBM System/360 Model 85 Integrated Operator's Console (220C). (Display Screen)
(6) IBM 2740 Communication Terminal Model I.

Console switching, the act of moving one console's capabilities to another console, can be done automatically, dynamically, and manually. Automatically switching to an alternate console occurs when the console is determined to be inoperative by the software. An operator command has been provided for dynamic console switching and console reconfiguration. The external interrupt key on the operator's panel provides manual switching to a new Master Console. The facility for the DISPLAY of the console configuration is provided through the *Display Consoles* operator command.

HARD COPY LOG, ROUTING CODES, AND TIME STAMPING: MCS provides the capability to have buffered or immediate hard copy. Specification of the hard copy device is provided at system definition and at system initialization. It eliminates the loss of information when graphic console operators delete messages and operator commands from their screen. Hard Copy collects and records all messages that have routing codes which intersect an assigned set. This set can be dynamically changed by the Master Console operator. Messages when sent to Hard Copy are prefixed by their routing codes and a time stamp from the system clock.

User Exit: An option is provided to enable the inclusion of a resident, user-written exit routine. This routine receives in a separate buffer a copy of each message before it is routed. Available to the routine are the following:

1. Message Text
   (read only)
2. Routing Codes
   (modify or suppress)
3. Descriptor Codes
   (modify or suppress)

This allows a user to impose his own functional routing mechanism.

These individual facts about MCS together form a multiple console environment which accomplishes some of the aforementioned 'desirable aspects.'

(a) Reliability—The extensive alternate chaining and automatic console switching combined with a wide variety of device types insures much greater dependability. Now one or more of the consoles can fail without a noticeable delay in system functioning.
(b) Communication Power—More consoles immediately increases message output capability by a factor almost equal to the number of consoles. The new devices supported such as the IBM 2250 and the IBM System/360 Model 85 Integrated Operator's Console 220C also deliver increased speed in terms of time between message issuance and appearance on the console.
(c) Auditability—The Hard Copy Log concept is a direct answer to this problem and affords a flexible means of recording the system's daily performance.
(d) Flexibility—The changing internal and external system environment can be coped with through the new operator's commands. The console configuration can be easily changed by bringing up new consoles or removing others from operation. Operating consoles can have their functional areas and command authorities changed. Flexibility on a shop level is given by the User Exit routine which allows a user to tailor a routing algorithm more in keeping with his own specific job types.

GROWTH IN THIS NETWORK

MCS clearly follows the lines of the solution. Although not a panacea for all the mentioned points it

does afford a basis upon which the remaining points could be obtained.

MCS routes messages through the use of predetermined codes attached to messages and consoles defined to receive specific message codes. This allows for splitting messages between the consoles but does not allow for messages that might be considered priority. Thus, if a volume of messages occurs the operator will attack the output serially and he may lose time responding to a number of less important messages before he gets to the priority message. MCS does provide, however, a mechanism for flagging messages that are considered needed action on the part of the operator, but still a priority message may be tucked at the end of the queue. This problem of priority messages will grow as the system continues to grow and the number of output messages and operator responses increase. Although priority routing is a valid point it should be noticed that a multiple console environment that properly uses functional routing minimizes problems in this area.

The security aspect has not been addressed during the MCS discussion. Certain abilities for security handling and monitoring exist through the USER EXIT and HARD COPY mechanism, and a routing code is assigned for security messages. Beyond this there is no defined mechanism for a security console to monitor system functions that may affect the security user. Since the master console always has the authority to issue all commands and other consoles may have limited command issuing authorities, either may inadvertently affect the security user by cancelling his job, halting his job, or varying system conditions. The security user should be able to monitor all system functions that may affect the status of his jobs.

MCS is device independent within the range of devices/consoles it presently supports and in some cases devices not presently supported can be hooked into an MCS environment if the user modifies internal restrictions that disallow such devices. A mechanism should be provided to allow total device independence. Thus, a user should be able to allow for a desired number of consoles, and later, say at system initialization time, decide what devices he wants to use as consoles.

Message standardization perhaps does not fall within the range of implementing a multiple console concept. However, standardizing message formats and text would have great impact on an MCS environment since the USER EXIT could have more effective text analysis. This would indeed make available more means of routing to the individual user. For example, MCS puts the job name on each message. Hence the USER EXIT could feasibly route by job name by incorporating a standard naming technique within the given shop.

OTHER PHILOSOPHIES

The last two sections dealt with one specific case, namely MCS, which should be recognized as just one example. It is not meant to be representative since any application is a function more of its environment than any other single factor and consequently cannot be considered a general solution. It (MCS) does however point out that more can be derived from a multiple console environment than meets the eye. The extremely obvious and trivial concept of more than one console, can play a significant role in large system enhancement if implemented in a meaningful manner. The basis upon which messages are routed seems to be the major point of difference between philosophies. No one method can be labeled as the best since different environments lend themselves to different routing algorithms.

Some systems have more than one Direct Access, Tape, or Unit Record pool so that functional routing as afforded by MCS would result in needless messages at certain locations. A more suitable algorithm in this case might be to route by unit address. In this way, a console could request all information concerning certain devices. This method also makes implementation of security measures more feasible since any operations concerning a particular volume could be monitored by receiving all messages concerning the unit upon which the volume is mounted.

Another routing algorithm which arises from the user who wishes to monitor his own program is the routing by job class or partition. With this concept a console would receive all messages referring to a particular job class. By properly assigning classes then, one could go to the appropriate console to watch his own job being processed.

These algorithms are for the most part batch oriented. A time-sharing environment would place different demands on the communication network and proper implementation of multiple consoles no doubt implies different routing algorithms.

SUMMARY

Beneath the existing multi-console answer to computer communication network problems lies the more significant matter of meaningful implementation. Proper implementation cannot be stereotyped since it depends for the most part on the environment.

Hardware aspects of secure computing

by LEE M. MOLHO

System Development Corporation
Santa Monica, California

INTRODUCTION

It makes no sense to discuss software for privacy-preserving or secure time-shared computing without considering the hardware on which it is to run. Software access controls rely upon certain pieces of hardware. If these can go dead or be deliberately disabled without warning, then all that remains is false security.

This paper is about hardware aspects of controlled-access time-shared computing.* A detailed study was recently made of two pieces of hardware that are required for secure time-sharing on an IBM System 360 Model 50 computer: the storage protection system and the Problem/Supervisor state control system.[1] It uncovered over a hundred cases where a single hardware failure will compromise security without giving an alarm. Hazards of this kind, which are present in any computer hardware which supports software access controls, have been essentially eliminated in the SDC ADEPT–50 Time-Sharing System through techniques described herein.[2]

Analysis based on that work has clarified what avenues are available for subversion via hardware; they are outlined in this paper. A number of ways to fill these security gaps are then developed, including methods applicable to a variety of computers. Administrative policy considerations, problems in security certification of hardware, and hardware design considerations for secure time-shared computing also receive comment.

FAILURE, SUBVERSION, AND SECURITY

Two types of security problem can be found in computer hardware. One is the problem of hardware failure. This includes not only computer logic that fails by itself, but also miswiring and faulty hardware caused by improper maintenance ("Customer Engineer") activity, including CE errors in making field-installable engineering changes.

The other security problem is the cloak-and-dagger question of the susceptibility of hardware to subversion by unauthorized persons. Can trivial hardware changes jeopardize a secure computing facility even if the software remains completely pure? This problem and the hardware failure problem, which will be considered in depth, are related.

Weak points for logic failure

Previous work involved an investigation of portions of the 360/50 hardware.[1] Its primary objective was to pinpoint single-failure problem locations. The question was asked, "If this element fails, will hardware required for secure computing go dead without giving an alarm?" A total of 99 single-failure hazards were found in the 360/50 storage protection hardware; they produce a variety of system effects. Three such logic elements were found in the simpler Problem/Supervisor state (PSW bit 15) logic. A failure in this logic would cause the 360/50 to always operate in the Supervisor state.

An assumption was made in finding single-failure logic problems which at first may seem more restrictive than it really is: A failure is defined as having occurred if the output of a logic element remains in an invalid state based on the states of its inputs. Other failure modes certainly exist for logic elements, but they reduce to this case as follows: (1) an intermittent logic element meets this criterion, but only part of the time; (2) a shorted or open input will cause an invalid output state at least part of the time; (3) a logic element which exhibits excessive signal delay will appear to have an invalid output state for some time after any input transition; (4) an output wire which has been con-

*The relationship between "security" and "privacy" has been discussed elsewhere.[3,4] In this paper "security" is used to cover controlled-access computing in general.

nected to an improper location will have an invalid output state based on its inputs at least part of the time; such a connection may also have permanently damaged the element, making its output independent of its input. It should be noted that failure possibilities were counted; for those relatively few cases where a security problem is caused whether the element gets stuck in "high" or in "low" state, two possibilities were counted.

A situation was frequently encountered which is considered in a general way in the following section, but which is touched upon here. Many more logic elements besides those tallied would cause the storage protection hardware to go dead if they failed, but fortunately (from a security viewpoint) their failure would cause some other essential part of the 360/50 to fail, leading to an overall system crash. "Failure detection by faulty system operation" keeps many logic elements from becoming security problems.

Circumventing logic failure

Providing redundant logic is a reasonable first suggestion as a means of eliminating single failures as security problems. However, redundancy has some limits which are not apparent until a close look is taken at the areas of security concern within the Central Processing Unit (CPU). Security problems are really in control logic, such as the logic activated by a storage protect violation signal, rather than in multi-bit data paths, where redundancy in the form of error-detecting and error-correcting codes is often useful. Indeed, the 360/50 CPU already uses an error-detecting code extensively, since parity checks are made on many multi-bit paths within it.

Effective use of redundant logic presents another problem. One must fully understand the system as it stands to know what needs to be added. Putting it another way, full hardware certification must take place before redundancy can be added (or appreciated, if the manufacturer claims it is there to begin with).

Lastly, some areas of hardware do not lend themselves too easily to redundancy: There can be only one address at a time to the Read-Only-Storage (ROS) unit whose microprograms control the 360/50 CPU.[5,6] One could, of course, use such a scheme as triple-modular redundancy on all control paths, providing three copies of ROS in the bargain. The result of such an approach would not be much like a 360/50.

Redundancy has a specialized, supplementary application in conjunction with hardware certification. After the process of certification reveals which logic elements can be checked by software at low overhead, redundant logic may be added to take care of the remainder. A good example is found in the storage protection logic. Eleven failure possibilities exist where protection interrupts would cause an incorrect microprogram branch upon failure. These failure possibilities arise in part from the logic elements driven by one control signal line. This signal could be provided redundantly to make the hardware secure.

Software tests provide another way to eliminate hardware failure as a security problem. Code can be written which should cause a protection or privileged-operation interrupt; to pass the test the interrupt must react appropriately. Such software must interface the operating system software for scheduling and storage-protect lock alteration, but must execute in Problem state to perform its tests. There is clearly a tradeoff between system overhead and rate of testing. As previously mentioned, hardware certification must be performed to ascertain what hardware can be checked by software tests, and how to check it.

Software testing of critical hardware is a simple and reasonable approach, given hardware certification; it is closely related to a larger problem, that of testing for software holes with software. Software testing of hardware, added to the SDC ADEPT–50 Time-Sharing System, has eliminated over 85 percent of present single-failure hazards in the 360/50 CPU.

Microprogramming could also be put to work to combat failure problems. A microprogrammed routine could be included in ROS which would automatically test critical hardware, taking immediate action if the test were not passed. Such a microprogram could either be in the form of an executable instruction (e.g., TEST PROTECTION), or could be automatic, as part of the timer-update sequence, for example.

A microprogrammed test would have much lower overhead than an equivalent software test performed at the same rate; if automatic, it would test even in the middle of user-program execution. A preliminary design of a storage-protection test that would be exercised every timer update time (60 times per second) indicated an overhead of only 0.015 percent (150 test cycles for every million ROS cycles). Of even greater significance is that microprogrammed testing is specifiable. A hardware vendor can be given the burden of proof of showing that the tests are complete; the vendor would have to take the testing requirement into account in design. The process of hardware certification could be reduced to a design review of vendor tests if this approach were taken.

Retrofitting microprogrammed testing in a 360/50 would not involve extensive hardware changes, but some changes would have to be made. Testing microprograms would have to be written by the manu-

facturer; new ROS storage elements would have to be fabricated. A small amount of logic and a large amount of documentation would also have to be changed.

Logic failure can be totally eliminated as a security problem in computer hardware by these methods. A finite effort and minor overhead are required; what logic is secured depends upon the approach taken. If microprogram or software functional testing is used, miswiring and dead hardware caused by CE errors will also be discovered.

Subversion techniques

It is worthwhile to take the position of a would-be system subverter, and proceed to look at the easiest and best ways of using the 360/50 to steal files from unsuspecting users. What hardware changes would have to be made to gain access to protected core memory or to enter the Supervisor state?

Fixed changes to eliminate hardware features are obvious enough; just remove the wire that carries the signal to set PSW bit 15, for example. But such changes are physically identical to hardware failures, since something is permanently wrong. As any functional testing for dead hardware will discover a fixed change, a potential subverter must be more clever.

In ADEPT–50, a user is swapped in periodically for a brief length of time (a "quantum"). During his quantum, a user can have access to the 360/50 at the machine-language level; no interpretive program comes between the user and his program unless, of course, he requests it. Thus, a clever subverter might seek to add some hardware logic to the CPU which would look for, say, a particular rather unusual sequence of two instructions in a program. Should that sequence appear, the added logic might disable storage protection for just a few dozen microseconds. Such a small "hole" in the hardware would be quite sufficient for the user to (1) access anyone's file; (2) cause a system crash; (3) modify anyone's file.

User-controllable changes could be implemented in many ways, with many modes of control and action besides this example (which was, however, one of the more effective schemes contemplated). Countermeasures to such controllable changes will be considered below, along with ways in which a subverter might try to anticipate countermeasures.

Countermeasures to subversion

As implied earlier, anyone who has sufficient access to the CPU to install his own "design changes" in the hardware is likely to put in a controllable change, since a fixed change would be discovered by even a simple software test infrequently performed. A user-controllable change, on the other hand would not be discovered by tests outside the user's quantum, and would be hard to discover even within it, as will become obvious.

The automatic microprogrammed test previously discussed would have a low probability of discovering a user-controllable hardware change. Consider an attempt by a user to replace his log-in number with the log-in number of the person whose file he wants to steal. He must execute a MOVE CHARACTERS instruction of length 12 to do this, requiring only about 31 microseconds for the 360/50 CPU to perform. A microprogrammed test occurring at timer interrupts—once each 16 milliseconds—would have a low probability of discovering such a brief security breach. Increasing the test rate, though it raises the probability, raises the overhead correspondingly. A test occurring at 16 *micro*second intervals, for example, represents a 15 percent overhead.

A reasonable question is whether a software test might do a better job of spotting user-controllable hardware changes. One would approach this task by attempting to discover changes with tests inserted in user programs in an undetectable fashion. One typical method would do this by inserting invisible breakpoints into the user's instruction stream; when they were encountered during the user's quantum, a software test of storage protection and PSW bit 15 would be performed.

A software test of this type could be written, and as will be discussed, such a software test would be difficult for a subverter to circumvent. Nevertheless, the drawbacks of this software test are severe. Reentrant code is required so that the software test can know (1) the location of the instruction stream, and (2) that no instructions are hidden in data areas. Requiring reentrant programs would in turn require minor changes to the ADEPT–50 Jovial compiler and major changes to the F-level Assembler. A small microprogram change would even be required, so that software could sense the difference between a fetch-protect interrupt and an execute-protect interrupt. Changes would be required to the ADEPT–50 SERVIS, INTRUP, DEBUG, and SKED modules. Were such a software test implemented, run-time overhead would likely be rather high for frequent breakpoint-insertions, since each breakpoint inserted would require execution of 50 or more instructions at run time. Lastly, programmers might not always like having to write reentrant code.

These implementation problems, though described for one specific test approach, are likely to extend to any test which attempts to interrupt running user pro-

grams via software, since any such test must isolate and augment every user's instruction stream.

Although the above does not demonstrate that detecting user controllable subversion hardware is impossible, it should indicate that by no means is such detection trivial. It is even harder than the above may suggest, for, as the next section indicates, while the subverter is busily building his controllable changes into the CPU he may add insult to injury by inactivating anti-failure and anti-subversion hardware.

Defeat of countermeasures

Hardware countermeasures that are easy to install are easy to remove. Redundant logic can be disabled with the same wire cutters or wire-unwrapper as other logic. This is also true of state-testing and parity-check logic. Alternatively, a clever subverter may add logic that picks up a test signal and uses it to inactivate his own logic while the test is performed. The subverter may, alternatively, arrange things so that tests always pass during his quantum.

Microprogrammed countermeasures are somewhat more difficult for the subverter to inactivate, since he must decode particular ROS addresses to know that a microprogrammed test routine has begun. Once he has done this, though, he may proceed as if he were dealing with hardware, since on a short-term basis, microprogram contents and addresses do not change. The net result of putting countermeasures in ROS rather than in logic is to raise the "work function" for subversion; that is, the subverter must do more work to achieve his goal, but he can reach it.

On the other hand, a thorough software test, such as the breakpoint test previously described, would be nearly impossible to defeat by hardware techniques. This is because of the transient nature of software: a subverter would have to decode program variables such as addresses via hardware; then even minor changes in the breakpoint-placing routines would make that hardware useless. One must recall, however, the large implementation and overhead problems inherent in a user-interrupting software test. In summary, countermeasures can be devised which have a high "work function," but they entail major costs in implementation and system efficiency.

Two assumptions have been inherent in this discussion; namely, that the subverter has both knowledge of system hardware (including subversion countermeasures) and means of changing the hardware. This need not be the case, but whether it is depends on administrative rather than technical considerations. Administrative considerations are the next subject.

Administrative policy

Special handling of hardware documentation and engineering changes may be worthwhile when commercial lines of computers are used for secure time-sharing. First, if hardware or microprograms have been added to the computer to test for failures and subversion attempts, the details of the tests should not be obtainable from the computer manufacturer's worldwide network of sales representatives. The fact that testing is done and the technical details of that testing would seem to be legitimate security objects, since a subverter can neutralize testing only if he knows of it. Classification of those documents which relate to testing is a policy question which should be considered. Likewise, redundant hardware, such as a second copy of the PSW bit 15 logic, might be included in the same category.

The second area is that of change control. Presumably the "Customer Engineer" (CE) personnel who perform engineering changes have clearances allowing them access to the hardware, but what about the technical documents which tell them what to do? A clever subverter could easily alter an engineering-change wire list to include his modifications, or could send spurious change documentation. A CE would then unwittingly install the subverter's "engineering change." Since it is asking too much to expect a CE to understand on a wire-by-wire basis each change he performs, some new step is necessary if one wants to be sure that engineering changes are made for technical reasons only. In other words, the computer manufacturer's engineering changes are security objects in the sense that their integrity must be guaranteed. Special paths of transmittal and post-installation verification by the manufacturer might be an adequate way to secure engineering changes; there are undoubtedly other ways. It is clear that a problem exists.

Finally, it should be noted that the 360/50 ROS storage elements, or any equivalent parts of another manufacturer's hardware that contain all system microprogramming, ought to be treated in a special manner, such as physically sealing them in place as part of hardware certification. New storage elements containing engineering changes are security objects of even higher order than regular engineering-change documents, and should be handled accordingly, from their manufacture through their installation.

GENERALIZATIONS AND CONCLUSIONS

Some general points about hardware design that relate to secure time-sharing and some short-range and long-range conclusions are the topics of this section.

Fail-secure vs. fail-soft hardware

Television programs, novels, and motion pictures have made it well known that if something is "fail-safe," it doesn't blow up when it fails. In the same vein, designers of high-reliability computers coined the term "fail-soft" to describe a machine that degrades its performance when a failure occurs, instead of becoming completely useless. It is now proposed to add another term to this family: "Fail-secure: to protect secure information regardless of failure."

The ability to detect failures is a prerequisite for fail-secure operation. However, all system provisions for corrective action based on failure detection must be carefully designed, particularly when hardware failure correction is involved. Two cases were recently described wherein a conflict arose between hardware and software that had been included to circumvent failures.* Automatic correction hardware could likewise mask problems which should be brought to the attention of the System Security Officer via security software.

Clearly, something between the extremes of system crash and silent automatic correction should occur when hardware fails. Definition of what *does* happen upon failure of critical hardware should be a design requirement for fail-secure time-sharing systems. Fail-soft computers are not likely to be fail-secure computers, nor vice versa, unless software and hardware have been designed with both concepts in mind.

Failure detection by faulty system operation

Computer hardware logic can be grouped by the system operation or operations it helps perform. Some logic—for example, the clock distribution logic—helps perform only one system operation. Other logic—such as the read-only storage address logic in the 360/50—helps perform many system operations, from floating point multiplication to memory protection interrupt handling. When logic is needed by more than one system operation, it is cross-checked for proper performance: Should an element needed for system operations A and B fail, the failure of system operation B would indicate the malfunction of this portion of operation A's logic.

Such interdependence is quite useful in a fail-secure system, as it allows failures to be detected by faulty system operation—a seemingly inelegant error detection mechanism, yet one which requires neither software nor hardware overhead. Some ideas on its uses and limitations follow.

The result of a hardware logic failure can usually be defined in terms of what happens to the system operations associated with the dead hardware. Some logic failure modes are detectable, because they make logic elements downstream misperform unrelated system operations. Analysis will also reveal failure modes which spoil only the system operation which they help perform. These failures must be detected in some other way. There are also, but more rarely, cases where a hardware failure may lead to an operation failure that is not obvious. In the 360/50, a failure could cause skipping of a segment of a control microprogram that wasn't really needed on that cycle. Such failures are not detectable by faulty system operation at least part of the time.

Advantage may be taken of this failure-detection technique in certifying hardware to be fail-secure as well as in original hardware design. In general, the more interdependencies existing among chunks of logic, the more likely are failures to produce faulty system operation. For example, in many places in a computer one finds situations as sketched in Figure 1. Therein,

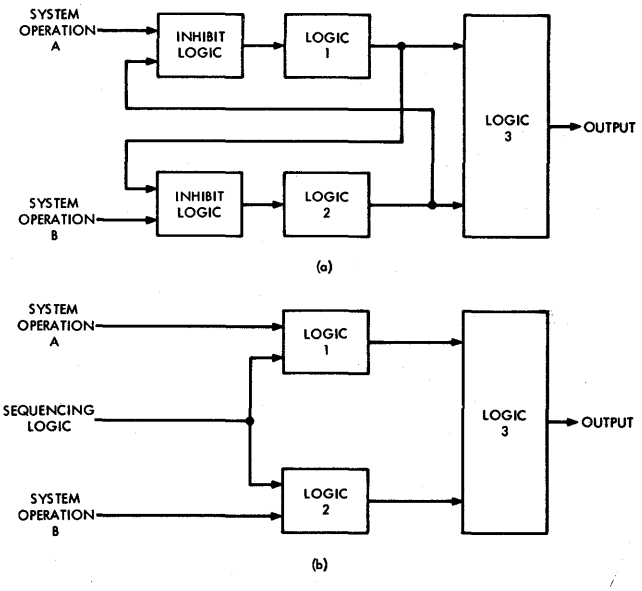


Figure 1—Inhibit logic vs sequencing logic

*At the "Workshop on Hardware–Software Interaction for System Reliability and Recovery in Fault-Tolerant Computers," held July 14–15, 1969 at Pacific Palisades, California, J. W. Herndon of Bell Telephone Labs reported that a problem had arisen in a developmental version of Bell's "Electronic Switching System." It seems that an elaborate setup of relays would begin reconfiguring a bad communications channel at the same time that software in ESS was trying to find out what was wrong. R. F. Thomas, Jr. of the Los Alamos Scientific Laboratory, having had a similar problem with a self-checking data acquisition system, agreed with Herndon that hardware is not clever enough to know what to do about system failures; software failure correction approaches are preferable.

TABLE 1—Control Signal Error Detection by Odd Parity Check on Odd-Length Data Field

DATA BITS *012 P*	*MEANING*
000 0	data error or control logic error*
000 1	0
001 0	1
001 1	data error
010 0	2
010 1	data error
011 0	data error
011 1	3
100 0	4
100 1	data error
101 0	data error
101 1	5
110 0	data error
110 1	6
111 0	7
111 1	data error or control logic error**

*Control logic incorrectly set all bits to zero.
**Control logic incorrectly set all bits to one.

System Operation A needs the services of Logic Group 1 and Logic Group 3, while System Operation B needs Logic Group 2 and Logic Group 3. Note at this point that, as above, if System Operation A doesn't work because of a failure in Logic Group 3, we have concurrently detected a failure in the logic supporting System Operation B.

A further point is made in Figure 1. Often System Operations A and B must be mutually exclusive; hardware must be added to prevent simultaneous activation of A and B. Two basic design approaches may be taken to solve this problem. An "inhibiting" scheme may be used, wherein logic is added that inhibits Logic Group 1 when Logic Group 2 is active, and vice versa. This approach is illustrated by Figure 1(a). Alternatively, a "sequencing" scheme may be used, wherein logic not directly involved with 1 or 2—such as system clock, mode selection logic, or a status register—defines when A and B are to be active. This approach is illustrated by Figure 1(b).

Now, "inhibit" logic belongs to a particular System Operation, for its function is to asynchronously, on demand, condition the hardware to perform that System Operation. It depends on nothing else; if it fails by going permanently inactive, only its System Operation is affected, and no alarm is given. On the other hand, "sequencing" logic feeds many areas of the machine; its failure is highly likely to be detected by faulty system operation.

A further point can be made here which may be somewhat controversial: that an overabundance of "inhibit"-type asynchronous logic is a good indicator of sloppy design or bad design coordination. While a certain amount must exist to deal with asynchronous pieces of hardware, often it is put in to "patch" problems that no one realized were there till system checkout time. Evidence of such design may suggest more thorough scrutiny is desirable.

System Operations can be grouped by their frequency of occurrence: some operations are needed every CPU cycle, some when the programmer requests them, some only during maintenance, and so on. Thus, some logic which appears to provide a cross-check on other logic may not do so frequently or predictably enough to satisfy certification requirements.

To sum up, the fact that a system crashes when a hardware failure occurs, rather than "failing soft" by continuing to run without the dead hardware, may be a blessing in disguise. If fail-soft operation encompasses hardware that is needed for continued security, such as the memory protection hardware, fail-soft operation is not fail-secure.

Data checking and control signal errors

Control signals which direct data transfers will often be checked by logic that was put in only to verify data purity. The nature and extent of this checking is dependent on the error-detection code used and upon the length of the data field (excluding check bits).

What happens is that if logic fails which controls a data path and its check bits, the data will be forced to either all zeros or all ones. If one or both of these cases is illegal, the control logic error will be detected when the data is checked. (Extensive parity checking on the 360/50 CPU results in much control logic failure detection capability therein.) Table 1 demonstrates an example of this effect; Table 2 describes the conditions for which it exists for the common parity check.

TABLE 2—Control Signal Error Detection by Parity Checking

DATA FIELD LENGTH:	PARITY:	CONTROL LOGIC ERROR CAUSES: all zeros	all ones
even	odd	CAUGHT	MISSED
even	even	MISSED	CAUGHT
odd	odd	CAUGHT	CAUGHT
odd	even	MISSED	MISSED

CONCLUSIONS

From a short-range viewpoint, 360/50 CPU hardware has some weak spots in it but no holes, as far as secure time-sharing is concerned. Furthermore, the weak spots can be reinforced with little expense. Several alternatives in this regard have been described.

From a longer-range viewpoint, anyone who contemplates specifying a requirement for hardware certification should know what such an effort involves. As reference, some notes are appropriate as to what it took to examine the 360/50 memory protection system to the level required for meaningful hardware certification. The writer first obtained several publications which describe the system. Having read these, the writer obtained the logic diagrams, went to the beginning points of several operations, and traced logic forward. Signals entering a point were traced backward until logic was found which would definitely cause faulty machine operation outside the protection system if it failed. During this tedious process, discrepancies arose between what had been read and what the logic diagrams appeared to show. Some discrepancies were resolved by further study; some were accounted for by special features on the SDC 360/50; some remain.

After logic tracing, the entire protection system was sketched out on eight $8\frac{1}{2} \times 11$ pages. This drawing proved to be extremely valuable for improving the writer's understanding, and enabled failure-mode charting that would have been intractable by manual means from the manufacturer's logic diagrams.

For certifying hardware, documentation quality and currentness is certainly a problem. The manufacturer's publications alone are necessary but definitely not sufficient, because of version differences, errors, oversimplifications, and insufficient detail. Both these and machine logic diagrams are needed.

Though the hardware certification outlook is bleak, an alternative does exist: testing. As previously described, it is possible to require inclusion of low-overhead functional testing of critical hardware in a secure computing system. The testing techniques, whether embedded in hardware, microprograms, or software, could be put under security control if some protection against hardware subversion is desired. Furthermore, administrative security control procedures should extend to "Customer Engineer" activity and to engineering change documentation to the extent necessary to insure that hardware changes are made for technical reasons only.

Careful control of access to computer-based information is, and ought to be, of general concern today. Access controls in a secure time-sharing system such as ADEPT–50 are based on hardware features.[7] The latter deserve scrutiny.

REFERENCES

1 L MOLHO
Hardware reliability study
SDC N-(L)-24276/126/00 December 1969

2 R LINDE C WEISSMAN C FOX
The ADEPT-50 time-sharing system
Proceedings of the Fall Joint Computer Conference Vol 35 p 39–50 1969
Also issued as SDC document SP-3344

3 W H WARE
Security and privacy in computer systems
Proceedings of the Spring Joint Computer Conference Vol 30 p 279–282 1967

4 W H WARE
Security and privacy: Similarities and differences
Proceedings of the Spring Joint Computer Conference Vol 30 p 287–290 1967

5 S G TUCKER
Microprogram control for system/360
IBM Systems Journal Vol 6 No 4 p 222–241 1967

6 G C VANDLING D E WALDECKER
The microprogram control technique for digital logic design
Computer Design Vol 8 No 8 p 44–51 August 1969

7 C WEISSMAN
Security controls in the ADEPT-50 time-sharing system
Proceedings of the Fall Joint Computer Conference Vol 35 p 119–133 1969
Also issued as SDC document SP-3342

TICKETRON—A successfully operating system without an operating system

by HARVEY DUBNER and JOSEPH ABATE

Computer Applications Incorporated
New York, New York

INTRODUCTION

In recent years, industry has witnessed the proliferation of complex on-line systems. More and more, computer management is recognizing the need to employ scientific methods to assist in the complex tasks of hardware/software selection and evaluation. This is especially true for real-time computer systems. As is well known, the distinguishing feature of real-time systems is that they are prone to the most spectacular failures ever witnessed in the computer industry. In many installations, real-time systems have become "hard-time" systems. The specter of potential failure has caused users to realize the importance of designing first, installing later. The sophisticated user has become aware of the fact that the rules of thumb and intuition that adequately described simple batch-type systems do not suffice when one is concerned with real-time systems. Real-time automation demands a certain amount of expertise on the part of the designer and implementor. In fact, systems which have been installed without adequate pre-analysis, more often than not, wind up with:

- Too expensive a central processor
- Too many ancillary components
- The wrong number of I/O channels
- Too elaborate a Supervisory System
- Poor communications interface

Clearly, the salient point we wish to establish is that real-time systems have a tendency to cost far in excess than necessary. Typically, the inefficient use of hardware is the staggering cost factor which most dramatically degrades the performance per dollar of a real-time system. Of course, our concern of performance per dollar would be an academic issue if the effect of improper design were to cause increases in hardware costs of the order of 10%. However, we maintain that the situation is much more drastic and such systems suffer excessive hardware costs in the order of 100%.

The reason for this state of affairs is that the design of real-time systems is not an art but rather a scientific discipline.[1,2] One must bring analysis to bear on the problems. To be sure, it is not the purpose of this paper to give a full treatment of this discipline. Rather, it is the purpose of this paper to demonstrate certain techniques[3,4,5] and their application to a real-life system, TICKETRON.

TICKETRON is a real-time ticket reservation and distribution system for the entertainment industry. In many respects it resembles most other real-time systems, therefore, the discussions concerning this system are by no means unique to it. That is to say, the approach and attitudes developed in the design and implementation of TICKETRON represent our philosophy toward real-time systems in general. We believe that using a successful system such as TICKETRON as the vehicle for presenting our philosophy concerning real-time systems, adds substance to our arguments.

The ultimate aim of our arguments is the concern for maximum performance per dollar of a system. TICKETRON is successful because it did achieve excellent performance per dollar. Specifically, the "industry standard" for this type of system priced the central facility hardware at over $60,000 per month. Through proper design, TICKETRON was able to accomplish better performance for less than $30,000 per month.

At the heart of the problem is the frenzy associated with multiprogramming in real-time systems, causing the need for supervisory programs. There has been a tendency in the past few years to implement operating systems which are so elaborate that the amount of computer time used for message processing can be matched or exceeded by the amount of time required

by the supervisor to maintain the job flow. In addition to the exorbitant overhead in time, there is the extra hardware cost associated with the inordinately large amount of dedicated core storage required by such operating systems. Further, a typical, modern-day, operating system presents itself as a labyrinth to the user who is required to make his application programs function in the unfamiliar and complex environment of the operating system. In most instances, the added burden on the user to cope with this labyrinth during program development and debugging is so costly in terms of manpower effort that it would have been far cheaper for him to have avoided trying to take advantage of the "standard supervisory package."

In short, these problems reflect a major paradox associated with third generation computer systems: "How can an operating system that costs nothing be so expensive?"

At this point, the reader might feel that we have overstated our position. No doubt he is able to point to many systems having constraints such that they require an elaborate operating system. We agree. Certainly a large on-line system which must perform a multitude of tasks cannot function without a complex supervisory system. Our point, however, is that too often simple systems are designed as if they were complex systems.

To summarize our approach, we believe that simplicity is the keynote of a good system design. If there is no need to multiprogram, don't! This is why TICKETRON is a successfully operating system without an operating system!

It is the intent of this paper to put forth the system design story for TICKETRON. The second section presents the design and the third section explains the design. The fourth section analyzes the design.

SYSTEM OVERVIEW

In addition to giving a functional description of the system, it is the purpose of this section to broadly specify the architecture of the system.

To begin, what is TICKETRON? It is a fully computerized ticket reservation and distribution system offering actual printed tickets at remote terminals. In short, it provides access to box offices from remote locations. In that sense, TICKETRON is an extention of the box office. It was originally intended to sell tickets for the entertainment industry. However, today it is also selling train tickets for the Penn–Central Metroliner. The system is practical in any application which involves the issuing of tickets. Remote sales terminals are installed at high-traffic points such as shopping centers, department stores, etc., and, of course, at box offices. It is a nationwide service having separate systems, each serving a geographical area. There are three central facilities at present: New York, Chicago and Los Angeles. Each central facility can support almost 900 terminals which can accommodate sales of 50,000 tickets per hour without any difficulty, under certain conditions (see the fourth section).

A remote terminal consists of a dedicated keyboard, a ticket printer and a receive-only teletype. A customer desiring tickets approaches a remote station and makes an inquiry concerning the availability of a performance. The terminal operator interrogates the system via the dedicated keyboard and receives a response in seconds at the teletype. The teletype message indicates what seats are available, if any. Then, if the customer is pleased with the selection, the operator will cause the system to "sell" the seats. Within seconds the actual tickets are printed out by the ticket printer. These are real tickets and the customer pays for them as he would at a box office. Therefore, in a genuine sense the remote station is an extension of the box office. Direct-access to the total ticket inventory guarantees remote buyers the best available seats at time of purchase (this is done automatically by a seat selection program). In addition to selling tickets, the system provides certain key reports for management information and also accurate accounting of ticket sales.

TICKETRON is a typical real-time system in that it is composed of four major constituents:

(1) the remote terminals with their communications network
(2) the line controller and buffers
(3) the processing unit and associated core storage
(4) the auxiliary storage with its connecting data channels

Knowing that these functional elements are required in the system, one must then determine what hardware is best suited for the job. Hopefully, this selection should be made on a performance per dollar basis. In short, this is what systems design is all about.

In the third section, we discuss certain procedural concepts that we consider important for accomplishing an effective system design. Further, we present some findings obtained by executing these procedures for the TICKETRON system. The remainder of this section will be devoted to an overview of the hardware and software architecture of the system.

An important result is the actual hardware configuration that was decided upon for TICKETRON. It was found that the system should be dedicated solely to the on-line, real-time tasks required of it.

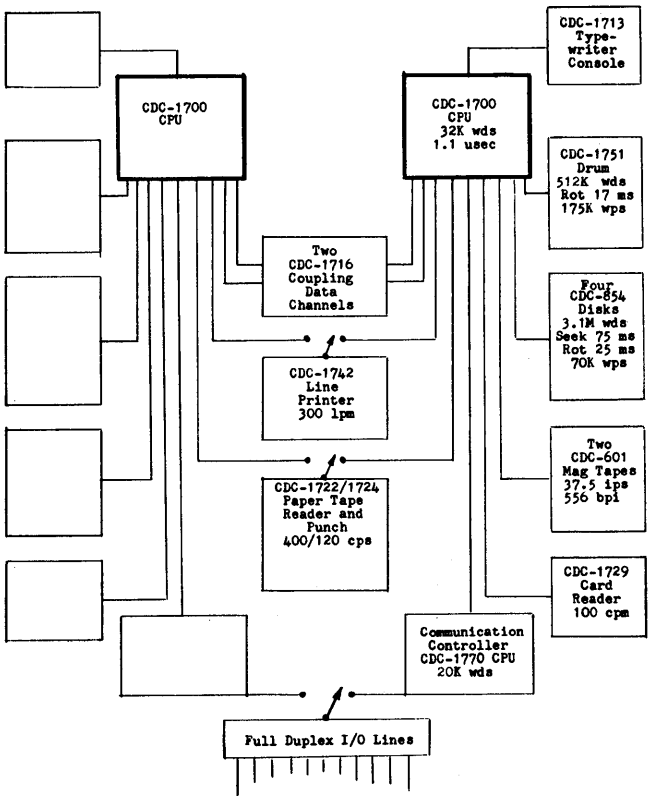


Figure 1—Central facility hardware configuration of TICKETRON

Further, it was found that the tasks were such that a process control type computer afforded the best performance per dollar in this situation. The computer system selected was a Control Data Corporation 1700. Figure 1 shows the central facility configuration. Reliability deemed it necessary that essential hardware be duplexed. The application is such that the system must be operative at certain critical times, for example, the peak ticket selling period just before a ballgame.

A result of the design shown in Figure 1 indicates that the TICKETRON system has two processors; one processor acts as a communications controller, while the other processes the messages. The front-end only handles the communication functions and contains the input and output line buffers. It does not examine the contents of the message. This last function is done by the message processing program which is resident in the central processor, which requests messages from the front-end. The communications program is resident in the front-end.

In addition to specifying the hardware, Figure 1 indicates the approximate characteristics of each device. The total monthly rental for the central facility hardware (including duplexing and maintenance) is about $30,000. We maintain that this is an achievement of understanding the characteristics of real-time systems and their design consequences.

As previously stated, the hardware configuration is a result of our design analysis. To be sure, the software design is not divorced from the performance analysis. In fact, one establishes certain programming considerations by analyzing their effect on the performance of the system. As a result, we decided on a single-thread program design. That is, at any one time there will be no more than one message in the system which is partially processed. There may be additional messages in the system, but these will be in one of two states; either awaiting processing (in input queue) or having completed processing (in output queue). We can use a single-thread programming concept because of the timings involved.

There are three major program modules: the communications program, the message processing program and an on-line utility program. The software design is such that each subroutine calls the next required subroutine. In essence, the system has one big program.

In considering the flow of a message through the system, we have the processing program receiving its messages from the input queue and after processing, delivering them to the output queue. The processing program is a single-thread program which deals with one message at a time. That is, it only accepts another message for processing after it has delivered one to the output queue. The processing program determines the next message to be processed as follows: it scans the input queue. When it finds a full buffer, it first checks the output queue to see if a buffer corresponding to that line is empty; if not, it will carry out the procedure for another line. If the processing program finds no candidates to be processed, it will then exit to what one might call a main scheduler program which does nothing but loop-the-loop. When the processing program has completed a message, the procedure is for it to loop back on itself. When it has work, it starts its cycle over again and does not return control to the main scheduler. That is, during a busy period, the processing program is continually looping; further, it is in complete charge since it has no open branches. In fact, during this time, the processing program has all the characteristics of an executive routine. However, in actuality the concept of "the executive" is foreign to this system.

The communications program, which is resident in the front-end processor, simply fills and empties the input and output buffers, respectively, for each communications line. The details of its operation are given in a later section, because the communication discipline was an integral part of the system design.

All programming on the system was done in FORTRAN because it offered the following advantages:

(a) Minimize programming costs and time.
(b) Machine independent.
(c) Partially self-documenting, no patching.
(d) Easy to modify.
(e) Easy communication between subroutines (and between programmers writing the subroutines).

History and evolution of the system

The TICKETRON system was conceived in January, 1967. A pilot system using a CDC 160A computer and modified teletype as ticket printer was used to demonstrate the feasibility of the system and to gain practical experience with automated ticket selling. The pilot ran from July to November, 1967.

Although TICKETRON has not varied much in concept over the years, the size and requirement of the system have undergone evolutionary changes. As a result, there have been three different computer equipment configurations.

1. Initially, TICKETRON was aimed primarily at the New York legitimate theater and advance sale for some sporting events, with 50 to 100 terminals selling 50,000 tickets per day. The first operational on-line configuration consisted of a CDC 1700 with 16K of core, 2 disks each with 3.1 million words, 2 tapes, a console teletype, and communications hardware sufficient for interfacing 16 voice grade (1200 baud) lines. This equipment was duplicated for reliability, with a printer and paper tape reader-punch added for off-line work. This system went operational in March, 1968.

2. It soon became apparent that sporting events were more important than originally anticipated, and that more terminals would be required. The number of lines was increased from 16 to 32 and the core increased from 16K to 32K. This allowed the system to handle over 500 terminals and well over 100,000 tickets per day. In addition, a drum with 256K words was added for high frequency files, in particular for the inventory needed for selling same day sporting events. This system became operational in January, 1969.

3. As terminals become more distant from the computer center, communication costs can be dramatically reduced by using communication concentrators. These city buffers are actually computers which perform "intelligent" functions such as formatting tickets. Since this required redesigning of the whole communications program, we took advantage of new technology and replaced the communication hardware by a computer front end with 20K of core at no increase in cost. This configuration can handle the equivalent of 56 phone lines and almost 900 terminals. Also, to accommodate more events two more disks are being added and the size of the drum increased to 512K words. This system will be operational in the spring of 1970. The analysis performed in this paper assumes this configuration.

SYSTEM DESIGN

We maintain that a successful system design is achieved by understanding the characteristics of real-time systems and their design consequences. The procedure for accomplishing this is essentially an iteration scheme:

(1) Specify a configuration; that is, define a prototype model of the system.
(2) Evaluate the configuration as to its operational characteristics. Essential to this step is a performance analysis which determines the capabilities and limitations of the prototype system.
(3) Make design recommendations on the basis of the evaluation.
(4) Are the recommendations substantial? If yes, continue. If no, end.
(5) Incorporate the recommendations by updating the system model. Then start again.

In this section, we shall present some results obtained by executing the above system design procedure for TICKETRON. At the heart of the procedure is the performance analysis.

TICKETRON typifies the operational characteristics of a real-time system. Namely, it is representative of a stochastic service system. The situation encountered is that the inputs to the system occur randomly and generate service requests of the central processor which are varied in type. In a poorly designed system, these random phenomena cause queueing and congestion problems. Therefore, a performance analysis which takes account of the random phenomena is essential to the design effort. The considerations of this approach manifests itself as follows: in a steady-state operation, the throughput of the system is defined as the average number of input messages to the system per second. Certainly then the throughput requirement is an important design criterion. Concurrently, with the throughput considerations is the requirement of a tolerable response time to each input message. In a given system, the situation usually encountered is that of having a desirably high throughput which, in turn, causes a system request queue to build up, thereby degrading response time. Therefore, pertinent to the system design is a knowledge of throughput versus

response time, which will provide the basis for a practical tradeoff.

In fact, the analysis of congestion forms the basis for design considerations of both the hardware and software; e.g., the analysis can answer basic questions such as "How many terminals can a particular system configuration support?"

Most systems design at present is based on the intuition and experience of the designer with plans to "check-out" the final system performance by simulation. Historically, this approach has led to poor systems. For example, if one determines storage requirements of a real-time system by a consideration of the average input loading, then the system would certainly fail during peak traffic loads. On the other hand, if one tried to design around the peak traffic, then one would have a system which is grossly over-designed. Obviously, an optimum system can only be achieved by a consideration of the entire loading distribution. That is, one must consider the traffic level that is not exceeded 90% of the time, 99% of the time, etc., in order to realize an efficient system design. Once such considerations are introduced, predictions essential to adequate design can be made accurately without the haphazard quality of intuition or reliance on previous experience which is not truly applicable.

Communications program

Because a TICKETRON facility must be able to support over 500 remote terminals, the nucleus of the system is the communication function. Therefore, most of the design effort was concentrated on the communication program. In fact, the success of TICKETRON is mostly attributable to its ability to efficiently handle so many terminals. We now briefly present some of the communications design.

To begin, the communication program administers the handling of message traffic to and from the system by performing the following functions.

1. To initiate polls by sequencing through a polling list which is dynamically maintained.
2. To transfer messages between the terminals and the main processing program and to do this in an efficient manner.
3. To maintain the physical status of terminals connected to the system.
4. To distinguish on message errors of a physical nature (as opposed to errors in message content).

The communications program is resident in, and executed by, a special processor as shown in Figure 1. This front-end processor is a CDC–1774 CPU, which is essentially a stripped-down CDC–1700 computer. It has 20K words of core memory available for the program and I/O line buffers. (In this system a computer storage word is composed of 18 bits: 16 data bits, 1 parity bit and 1 program protect bit.) Associated with each line is essentially four buffers, two for input and two for output. Actually, one of the input buffers is in an area which contains ten buffers shared by all the lines. As we shall see below, the system will only issue a poll on the line if one of the input buffers is available. That is, you only take a message in if you have room for it. As was previously discussed, the message processing program will only start processing a message if there is an output buffer available. Hence, at any one time the system generally will contain no more than four messages for a particular line. Therefore, because of this "throttling" effect, one need not program for, nor worry about the problems associated with excessive internal queueing.

Each line is full duplex and transmission is done asynchronously. The system outputs 9-bit characters (1 start bit, 6 data bits, 1 parity bit, 1 stop bit) at 1200 baud or 7.50 msec per character, and inputs 7-bit characters (1 start bit, 4 data bits, 1 parity bit, 1 stop bit) at 800 baud or 8.75 msec per character. The reason why only 4 data bits are required on the input side, is that all input messages are in the form of a restricted fixed format. The terminal input device is a dedicated keyboard with dedicated columns allowing the entry of such pertinent information as: event code, performance date and time, and certain seat qualifiers. Associated with this data is one of three function codes: inquiry, buy, or buy alternate. The buttons on the keyboard are such that they only permit one per column or function to be depressed at any one time. Therefore, every input message to the system is of fixed size, 19 characters. The advantages of this scheme as to programming and operation are obvious.

The communication program uses a polling technique that can uniquely address each terminal in the system. The poll message uses four characters. The system uses no "hand-shaking" characters such as ACK or NACK. A poll to a keyboard causes it to transmit if the transmit button is depressed. This is accomplished in about 200 milliseconds. If the transmit button is not depressed when it receives at poll, it sends no response. The communication program will allow the terminal 200 milliseconds to respond before it infers a non-acknowledgment.

The communication program will perform the following logic for each communication line on a periodic execution cycle.

1. Check disposition of the "receive" line buffer,

three possibilities exist: free, full, or busy (200 milliseconds have not elapsed since last poll).

2. If free, check if there is available space for a message in an input message buffer. If the space exists for an input message, then prepare a poll message for the next non-busy terminal on that line. Go to 5.

3. If full, check for transmission errors, then move the message to the input message queue (space will always be available). Once this is done, the "receive" line buffer is free, go to 2.

4. If busy, go to 5.

5. Check disposition of the "send" line buffer, two possibilities exist: free or busy.

6. If free, check for message in the output buffer for this line, if there is one, send it, otherwise done. (Start algorithm again at 1 for next line.)

7. If busy, done. (Start algorithm again at 1 for next line.)

We purposely did not clutter the algorithm with implementation details since it would only cause to mar the simplicity of the scheme. For instance, the output required for the printing of a ticket is transmitted in four segments. Interspersed between the segments may be poll, light and TTY messages for other terminals on the line. Further, it is clear that the scheme requires use of certain kept tables which reflect terminal and line activity.

The basic philosophy in the given design of the communication program has been to maintain the integrity of its true function. That function being, that it is simply an intermediary between the message processing program and the terminals. Speaking loosely, it should be synchronized with the actions of the message processing program. In fact, since it is ultimately the responsibility of the processing program to respond to the terminals, then the communication controller should only react to the needs of the message processing program. For example, if at any given time the message processing program has enough work, then there is no need for the communication program to poll terminals for the purpose of bringing in more messages. To do otherwise would be illogical.

PERFORMANCE ANALYSIS

In this section, we give the results of a queueing analysis of the system in order to determine its capabilities and limitations. As argued in the second section, response time versus throughput is the basis in terms of which to measure the performance of the system. The throughput capability of any system is determined by certain utilization factors.

Utilization factor is a well defined mathematical concept of queueing theory. Given a facility which has some random arrival pattern for requests such that the average input rate is λ arrivals per second, then let each arrival place a demand on (tie-up) the facility for some average time, T_s seconds. That is, during the time T_s (service time), the facility is not available to any other arrival. Then we have the utilization factor of the facility defined by

$$U = \lambda T_s \tag{1}$$

In short, it represents the percentage of time the facility is tied-up. Obviously, U should not exceed 100%!

For TICKETRON, the throughput is measured in terms of the number of tickets per hour that the system is capable of outputing. These determinations start with a specification of the input traffic to the system.

We distinguish two types of terminals: box office and remotes. Remotes print what we call full-tickets (308 ch of data). In addition, box office terminals are capable of also selling half-tickets (119 ch of data), which are useful for same-day events. Remotes can only buy tickets after an inquiry has been previously executed, whereas, a box office may execute a direct buy. The reason for this is that box office attendants are more familiar with their own inventory and therefore, have little need to make inquiries. The characteristics of remotes are such that they average 1½ inquiries for each buy transaction. In contrast, at a box office you have on the average about ⅔ of an inquiry per buy transaction. A buy transaction requires on the average the printing of three tickets. Table I gives a distribution of the number of tickets sold per transaction. Because most tickets are bought in pairs, the distribution is "tight" about the average of three, as verified by the small squared coefficient of variation

TABLE I—Distribution of Various Types of Ticket Sales
Average number of tickets sold per buy transaction equals 2.95, with a standard deviation of 1.72.

Number of tickets sold per buy transaction	*Distribution (Percent of occurrence)*
1	10%
2	49%
3	10%
4	18%
5	4%
6	4%
7	1%
8	2%
9	2%

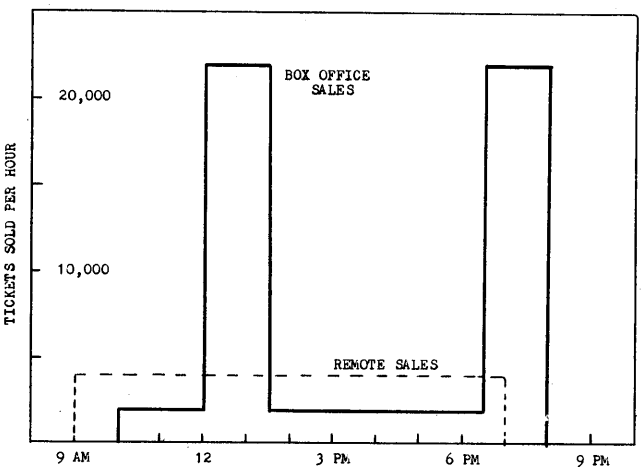


Figure 2—Histograms of hourly rate of ticket sales

which equals .34. Therefore, for calculational purposes, we may consider this a constant distribution and make use of the simple queueing formulas associated with constant service.

A typical operational day for TICKETRON is represented by sales of about 120,000 tickets, which is equivalent to 40,000 transactions. Unfortunately, this traffic is not distributed evenly throughout the day, hence, the system must be able to accommodate peak traffic loads. Figure 2 depicts histograms of the hourly rate of ticket sales, which we shall use to establish peak traffic loads. We note that the remote sales are evenly distributed at 4,000 tickets per hour over a ten hour day and they account for ⅓ the total load. Of these 4,000 tickets per hour, we estimate that one-tenth or 400 are for same-day events while 3,600 are for future events. In contrast, the box offices have sharp peaks for 1½ hour periods just before afternoon and evening performances. It is estimated that the box offices sell 2,000 tickets per hour for future events evenly over a ten hour period, which accounts for 20,000 tickets; whereas, the other 60,000 are for same-day events and are sold over a three hour period at an even rate of 20,000 per hour. These same-day events are usually sold as direct buys, hence, on the average we estimate that they cause only about .2 inquiries per sale. We note that since ⅔ of the sales average .2 inquiries per sale and ⅓ average 1.5 inquiries per sale, then on the average a box office has about ⅔ of an inquiry per buy transaction. In summary, during a peak hour, the box offices sell 22,000 tickets while the remotes sell 4,000.

Because most of the peak traffic represents sales for same-day events, the system keeps this inventory on high speed drums for fast retrieval. The processing service times for the various types of transactions are given in Table II. These timings are almost a constant independent of the number of tickets, therefore, we will assume them constant. The timings include all I/O times which are not overlapped with the processing, since the main processing program is a single-thread routine.

TABLE II—The Processing Service Times for Various Types of Transactions

Type of Transaction	*Processing Time in Seconds*	
	same-day events	*future events*
Inquiry	.135	.338
Buy following an Inquiry	.099	.248
Direct Buy	.180	.450

At this point, it is of interest to demonstrate how the processing times limit the throughput of the system. As argued in an earlier section, the inputs to the system are random, in fact we maintain that they generate a Poisson arrival stream to the processor. This phenomenon causes queueing of the inputs. Therefore, the total processing time or cpu response time must include waiting time for the cpu. The simple queueing formula which determines the average waiting time for a single-server queue with Poisson input rate λ, mean service time T_s, and second moment of the service time b_2, is given

$$W = \frac{U}{2(1-U)}\left(\frac{b_2}{T_s}\right) \tag{2}$$

where U is the utilization factor which is determined by eq. (1). To be sure, W becomes intolerably large as U approaches 100%, which is the limitation that governs the capability of the system. Hence, to obtain the cpu response time for a particular type of input, we just add its service time to the waiting time W. Let us now determine the average cpu response time for three different operational environments.

Case I. (Box office peak hour)

Assume for this case that all inputs to the system are direct buy transactions from box offices for same-day events. One may envision this situation to prevail for a period of about an hour on a day when every baseball team has an afternoon game and the remotes are closed. (Memorial Day is an example of such a day.) It is reasonable that in this environment there will be a

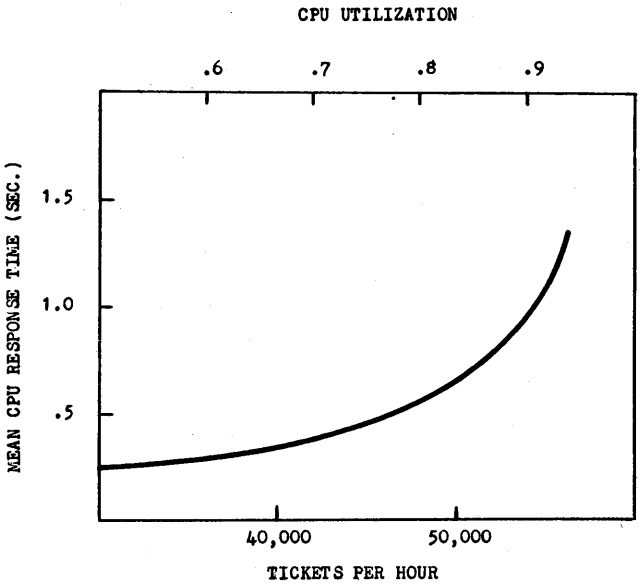


Figure 3—Average CPU response time for a direct buy transaction as a function of CPU utilization and tickets sold per hour for an input traffic generated only by box offices for same-day events, (Case I.)

negligible number of inquiries and selling for future events. From Table II, we have the cpu service time for this case as $T_s = .180$ seconds. Since the service time is constant, the second moments is equal to the mean-squared. Let λ equal the average number of input transactions per second or in this case, it is equal to average rate of buy transactions. Therefore, the cpu utilization and response time are given

$$U = \lambda(.180) \tag{3a}$$

$$T_{cpu} = \frac{U}{2(1-U)}(.180) + (.180) \tag{3b}$$

Figure 3 is a graph of this result. Because each buy transaction represents an average sale of three tickets and since there are 3,600 seconds in an hour, then $(10{,}800)\lambda$ represents the number of tickets sold per hour. This quantity is also given in Figure 3. We observe that in this environment the system can sell in the order of 50,000 tickets per hour.

Case II. (Remote peak hour)

In this case, let us assume that all the inputs to the system are generated by remote terminals and are sales for future events. This situation may very well occur on certain rare days which have very few events and during the time that box offices are usually not active. Since remote terminals cannot make a direct buy, the input transactions in this case will consist of inquiries and buys (only after an inquiry). Hence, there will be at least one inquiry for each buy, but in fact, we maintain that on the average there will be 1½ inquiries for each buy transaction. Therefore, if λ is the number of buys per second, then $(1.5\lambda + \lambda)$ equals the rate of input transactions to the system. From Table II, we calculate the mean cpu service time for a transaction to be $(.6)(.388) + (.4)(.248) = .302$ seconds. Hence, the cpu utilization in this case is given

$$U = (2.5\lambda)(.302) = \lambda(.755) \tag{4a}$$

As before, the number of tickets per hour equals $(10{,}800)\lambda$. The second moment of the service time equals .093 seconds–squared. Therefore, the average cpu response time to a buy transaction in this environment is given

$$T_{cpu} = \frac{U}{2(1-U)}\left(\frac{.093}{.302}\right) + (.248) \tag{4b}$$

Figure 4 is a graph of this result. We observe that in this environment the system can sell in the order of 10,000 tickets per hour.

It is interesting to note the contrast of this result with that given for the environment in Case I. We may consider these two cases as extremes to which the system can respond. Let us now turn to a study of the

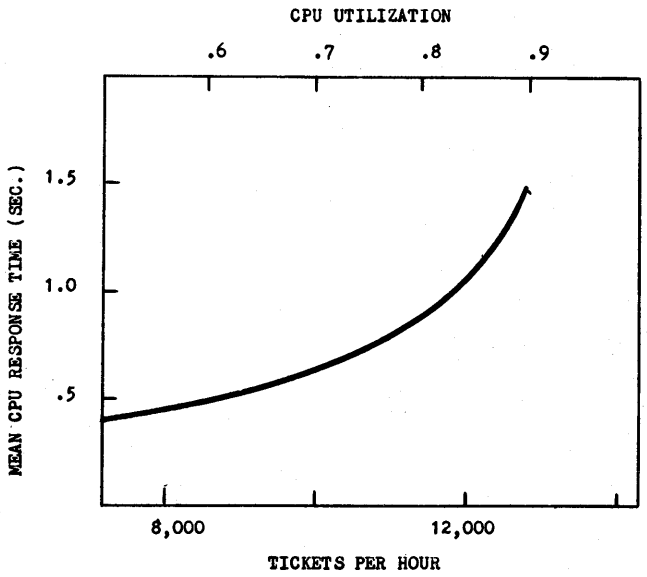


Figure 4—Average CPU response time for a buy transaction as a function of CPU utilization and tickets sold per hour for an input traffic generated only by remote terminals for future events, (Case II.)

TABLE III—Distribution of Transaction Types for a Realistic Peak Hour
λ is the average rate of buy transactions.

Type of Transaction	*Average Input Rates* *from box offices*	*from remotes*	*Processing Service Times in Seconds*
Same-day Events			
Direct Buy	$.8(20/26)\lambda$		.180
Inquiry	$.2(20/26)\lambda$	$1.5(.4/26)\lambda$	.135
Buy following an Inquiry	$.2(20/26)\lambda$	$1(.4/26)\lambda$	.099
Future Events			
Inquiry	$1.5(2/26)\lambda$	$1.5(3.6/26)\lambda$	.338
Buy following an Inquiry	$1(2/26)\lambda$	$1(3.6/26)\lambda$	.248

system which more closely corresponds to a realistic environment to which the system is subjected more often.

Case III. (Realistic Peak Hour)

Let us assume an input traffic mix as determined from a peak hour of the histograms given in Figure 2. Specification of this mix, as discussed above in this section, establishes the profile given in Table 3. Again, λ equals the average rate of buy transactions. We find that the total number of inputs per second is 1.5λ. Also, we find that the mean cpu service time for this traffic is .209 seconds. Therefore, the cpu utilization is given

$$U = (1.5\lambda)(.209) = \lambda(.314) \qquad (5a)$$

As before, the number of tickets per hour equals $(10{,}800)\lambda$. The second moment of the service time equals .050 seconds–squared. Therefore, the average cpu waiting time for processing in this environment is given

$$W_{cpu} = \frac{U}{2(1-U)}\left(\frac{.050}{.209}\right) \qquad (5b)$$

In Figure 5, we graph the following two functions: $W_{cpu} + .099$ and $W_{cpu} + .248$ which represent the average cpu response times to a buy following an inquiry for a same-day event and a future event, respectively.

Figure 5 shows that as far as cpu processing is concerned, the system can accommodate peak hour sales of 26,000 tickets in the environment specified by the histograms of Figure 2 which represents possible traffic distribution for sales of 120,000 tickets in a day. In a sense, this situation is to be expected because typically, the limitations for a system of this sort are not determined by the cpu processing capability but rather by the congestion of the communication lines. Further, since the output transmission dominates the communication load in such systems, it is the traffic on the output line and the number of such lines which truly govern the throughput capability of the system. For example, since the output transmission time required to print a full ticket is about 4 seconds, then the theoretical maximum that a line can output is 900 tickets per hour. Hence, if the system only had 16 lines, it could not achieve the throughput levels required. Therefore, due to these considerations, we shall now investigate the system in terms of the activity on a single line.

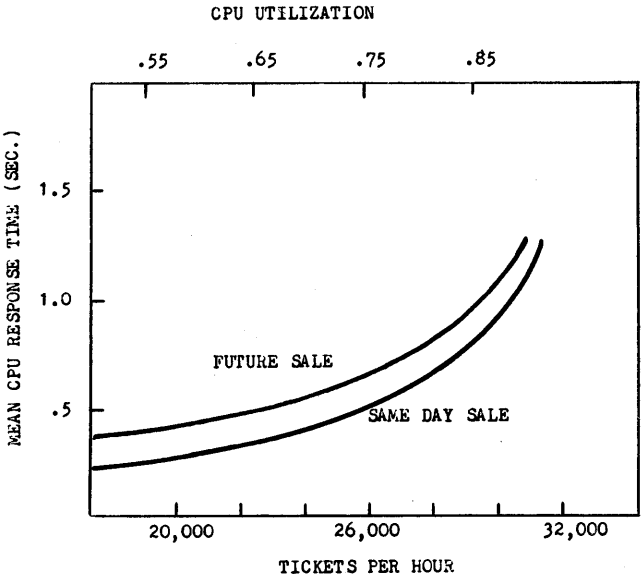


Figure 5—Average CPU response time for a buy following an inquiry as a function of CPU utilization and tickets sold per hour for the input traffic mix defined as "realistic" peak hour, (Case III.)

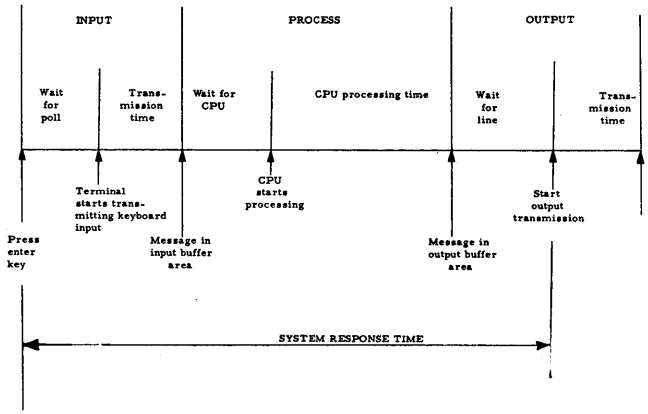


Figure 6—Timing diagram for the flow of a message through the system. Definition of system response time

System Response Time

We shall now determine the response time for a terminal on a specified communication line in terms of the output utilization and the output rate of tickets for that line. For this system, we shall define the response time as the elapsed time between the initiation of a request and the start of output data transmission. The timing schematic of Figure 6 illustrates this quantity. Further, the timing diagram depicts the delays encountered by the passage of a particular request through the system.

Hence, the system response time is the sum of four random variables: the waiting time for the terminal to be polled, plus the time required to transmit the input data, plus the cpu time required to process the input request, plus the waiting time for the output line to become free in order to commence transmission. The response time is given by

$$T = W_{\text{poll}} + T_{\text{in}} + T_{\text{cpu}} + W_{\text{out}} \qquad (6)$$

We have a system of three queues in tandem, input, processing and output. Essentially, the response time is figured by calculating the delays encountered at each queue and convolving the results. The procedure and assumptions required for such an analysis are given in reference.[3]

Unfortunately, in this analysis there is one slight complication; namely, the phenomenon of blocking for queues in tandem. For two queues in tandem, blocking occurs when a customer completes his current service but cannot move to the next queue because its limited waiting line is filled. Thus further service in the first queue is blocked until the second queue completes a service. If we recall the working of the communications program, it is such that polling is only performed if there is space in the input queue for a transaction, otherwise the system does not poll terminals for requests. That is, if at any given time the message processing program has enough work (the input queue is full), then there is no need for the communication program to poll terminals for the purpose of bringing in more messages, since to do so would serve no purpose.

This situation is unfortunate only for the analysis of the system, however, it is a real advantage in terms of system operation because it permits the system to optimize resource allocations concerning input message storage. Now because of buffer allocations, there are always enough transactions in core for processing and outputing, therefore, this blocking causes *no* forced idle time of the second and third queues. Hence, in the resulting system a request has the same total response time as it would have in a system where blocking of this sort were not present. In other words, if we were to calculate the individual terms on the right hand side of equation (6) assuming no blocking, the individual quantities would not represent the TICKETRON system, however, the total *does*, and is a valid representation for the response time of the TICKETRON system.

To this end, we now calculate the individual terms in equation (6) for a system which allows unlimited queues and continuous polling. We shall perform the calculation for two different configurations of output lines.

In the first configuration for an output line we shall assume that it has 12 remote terminals attached to it, all selling full tickets for future events. Further, we assume that the cpu is processing in the environment as specified in Case III. We want the response time to a buy transaction for a future event. Hence, for the cpu processing time we use equation (5b), giving $T_{\text{cpu}} = W_{\text{cpu}} + (.248)$. We assume that the throughput level of the cpu is at 26,000 tickets per hour, this corresponds to a cpu utilization of .756. Then from the appropriate graph in Figure 5, we find that for the environment being considered, $T_{\text{cpu}} = .62$ seconds.

Let us next calculate the delays encountered on inputing. First, we have the elapse time a terminal must wait in order to receive a poll once the transmit button on the keyboard has been depressed. We recall that for calculational purposes we consider a system with continuous polling. Therefore, we may envision that the 12 terminals on the line are being polled cyclically at the same rate. Because the pointer can be anywhere in the list when the terminal initiates a request for transmission, that particular terminal must wait on the average $N/2$ polling times before it receives a poll, where N is the number of terminals on the line. As dis-

cussed in Section 3, the communications program will successively poll terminals on a line every 200 milliseconds, provided the output line is free to send out a poll message. Therefore, when the line is free, the wait for a poll is $(N/2)(.2)$ seconds. On the other hand, when the output line is busy transmitting tickets, the polling rate is variable as follows. The transmission of a ticket is accomplished in four parts. The first operation is to send a short message which slews the ticket into the printer. The slew time is 1.2 seconds, however, during most of this time the line is free to permit polling at the normal rate. Hence, during slew time, we can send out five polls. Next, the data required to print the ticket is transmitted in three equal segments, and between each segment a poll message is sent (in addition to teletype and light messages). In the printing of a full ticket, each segment requires transmission of 103 data characters plus 4 control characters, which takes about .8 seconds. Therefore, to estimate the average time between polls during full ticket printing, we observe that we can send out 8 polls in about 4 seconds, which gives an average polling time of .5 seconds. Then when the line is busy, the wait for a poll is $(N/2)(.5)$ seconds.

Having found the average contribution to W_{poll} for free and busy conditions of the line, we now form a weighted sum to obtain the desired result. The weighting factors are expressed simply in terms of the utilization of the output line U_{out}. Since U_{out} is the average percent of time the line is busy, then $(1 - U_{\text{out}})$ is the average percent of time the line is free. Therefore

$$W_{\text{poll}} = (1 - U_{\text{out}})(N/2)(.2) + U_{\text{out}}(N/2)(.5) \quad (7a)$$

where, using $N = 12$ gives

$$W_{\text{poll}} = 1.2 + 1.8U_{\text{out}} \quad (7b)$$

Following the poll delay, is the time required for input transmission, T_{in}. As discussed earlier, the input message is of fixed size of 19 characters. At 8.75 msec. per ch., the time to transmit is .167 seconds, however, added to this is 70 milliseconds required to turn on the modem. Therefore, $T_{\text{in}} = .237$ seconds.

The final delay to be calculated is the waiting time encountered when the system is ready to transmit the data necessary to print tickets. Naturally W_{out} is a function of the line utilization and the service time required to print three tickets. As mentioned previously, we assume for calculational purposes that each buy transaction is for three tickets. We now determine the service time on the output line caused by such a transaction. First, there is the actual printing time for a ticket. This is equal to the transmission time since the printer prints at line speed of 1200 baud or 7.5 msec. per character. One full ticket requires 308 characters of data, plus 16 control characters, plus 1.2 seconds of slew time, which totals to 3.63 seconds. Therefore, three tickets take 10.89 seconds. However, this is not the only contribution to the traffic on the output line.

In addition to tickets, there are poll, teletype and light messages also being transmitted over the output line. As discussed above in connection with polling, the transmission of a ticket is segmented into four parts. The other messages are transmitted interspersed between the segments. Therefore, between each data segment we may figure on one poll message, one light and about two teletype messages being sent. Because the teletypes print at a much slower speed than 1200 baud, each teletype has a six character buffer. Hence, a teletype message is sent out in blocks of ten characters, six data plus four control characters. The teletype traffic accounts for the response to inquiries and audit trail associated with each transaction. We estimate that a teletype response to an inquiry requires 72 characters of data or 12 blocks of TTY messages, and an audit trail requires 12 characters of data or 2 blocks of TTY messages. Therefore, each buy transaction causes 20 blocks of TTY messages to be transmitted, since there are on the average 1½ inquiries per buy. Added to this traffic are nine poll messages and nine light messages each of four characters, for each buy transaction since the three tickets have nine data segments. (Note, that the poll messages sent out during slew time are overlapped.)

Hence, in addition to the ticket printing time for a buy transaction, we must add the transmission time for 272 characters which is 2.04 seconds to take account of the other activity on the output line. Therefore, the service time on the output line is 12.93 seconds for a buy transaction. If λ equals the average number of such transactions per second, then the utilization factor for the output line, $U_{\text{out}} = \lambda(12.93)$. As before, $(10{,}800)\lambda$ represents the number of tickets sold per hour over the line. Hence, the average waiting time in this case is given

$$W_{\text{out}} = \frac{U_{\text{out}}}{2(1 - U_{\text{out}})}(12.93) \quad (8)$$

Finally then, we accumulate these quantities to determine the response time as prescribed above using equation (6). We have that the average response time for this configuration is given by

$$T = (1.2 + 1.8U_{\text{out}}) + (.24) + (.62) + \left[\frac{U_{\text{out}}}{2(1 - U_{\text{out}})}(12.93)\right] \quad (9)$$

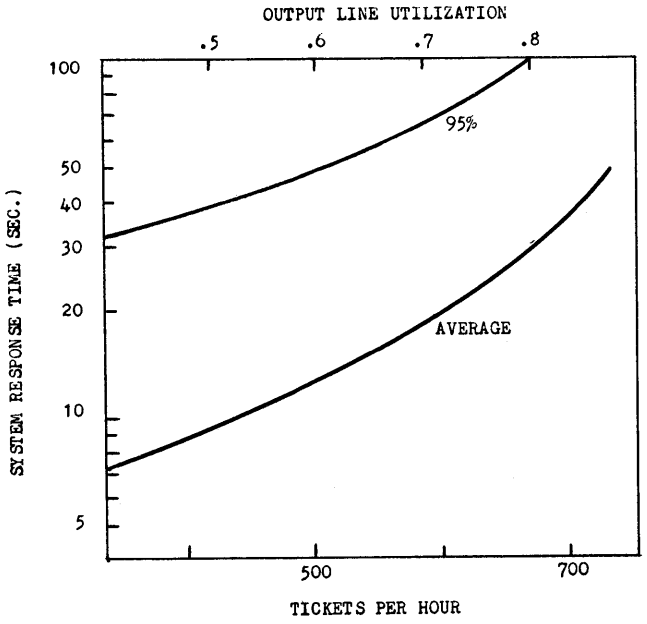


Figure 7—Average system response time for a buy transaction as experienced by a remote on an output line which has 12 remote terminals all selling full tickets. Also included is an estimate of the 95% system response time for this case; that is, the response time which is not exceeded by 95% of the transactions

Figure 7 includes a graph of this result. We observe the validity of the conjecture stated initially; namely, that the determination of the response time is dominated by the output transmission time. In other words, the contribution or congestion caused by the third queue (output) essentially determines the response time. Because of this, we may use a simple argument to estimate the 95th percentile of response time, that is, the response time which is not exceeded by 95% of the transactions. We argue that the system response time distribution is estimated by the waiting time distribution of the output queue which happens to be a queue with constant service time. For such a queue, we know the relationship that exists between the 95th percentile of waiting time and the average waiting time. Therefore, we use the same relationship to estimate the 95% system response time from the average value. A graph of this estimation is also given in Figure 7.

Having considered the case for an output line with 12 remote terminals all selling full tickets, let us now turn to what might be considered a more efficient situation. Assume an output line configuration with 2 box office terminals selling ½ tickets and 10 remote terminals selling full tickets. Again we have 12 terminals on the line, except in this case the box offices generate less line utilization per ticket sold, so that the throughput of the line is increased. Also, a box office terminal is polled twice as often as a remote terminal. (In fact, the polling sequence used by the system for this case is as follows: ···, R1, R2, R3, R4, R5, B1, B2, R6, R7, R8, R9, R10, B1, B2, R1, ··· where R designates remote and B designates box office.) Therefore, the box offices will have a better response time. Further, we assume that the ticket mix on the line is such that 50% of the sales are ½ tickets and 50% are full tickets. That is, the two box office terminals sell as much as the ten remotes since they are the only ones capable of selling ½ tickets.

We shall calculate the response time for a direct buy transaction as experienced by a box office. The calculations for this case are similar to those presented above. We find the response time given by

$$T = (.70 + 1.05U_1 + .18U_2) + (.24) + (.47) + \left[\frac{U_{ou}}{2(1 - U_{out})}\left(\frac{112.9}{10.3}\right)\right] \quad (10a)$$

where

$$U_{out} = \tfrac{1}{2}\lambda(12.93) + \tfrac{1}{2}\lambda(7.66) = U_1 + U_2 \quad (10b)$$

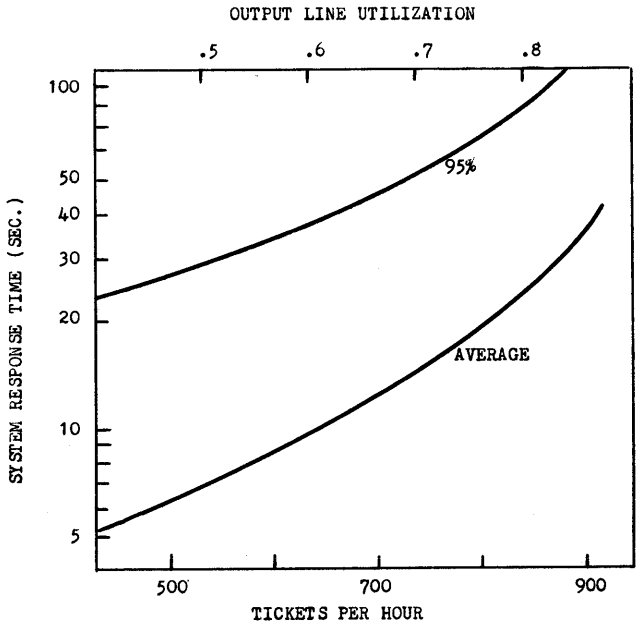


Figure 8—Average and 95% system response time for a direct buy transaction as experienced by a box office on an output line which has 2 box office terminals selling ½ tickets and 10 remote terminals selling full tickets. The ticket mix is such that 50% of the sales are ½ tickets and 50% of the sales are full tickets

That is, the output line utilization factor has two contributions, U_1 and U_2 which are the utilization factors due to transmission of full tickets and ½ tickets, respectively. Figure 8 depicts the average and 95% response time for this case. We observe that for an average response time of 10 seconds and a 95% response time of 40 seconds the throughput of the line is limited to 630 tickets per hour. Whereas, in the previous configuration of a remote selling full tickets, the same response time only afforded the line to output 430 tickets per hour. Therefore, the use of ½ tickets permits the system to sell more tickets per line. In fact, the more efficient configuration will allow the system to sell 26,000 tickets per hour using 41 lines such that the average response time is about 10 seconds.

REFERENCES

1 A WEINGARTEN
Analyzing a real-time system
Proc of ACM National Conference pp 179–182 1966

2 P H SEAMAN
Analysis of some queueing models in realtime systems
IBM Manual No F20-0007 IBM Corporation Poughkeepsie New York 1965

3 J ABATE H DUBNER S WEINBERG
Queueing analysis of the IBM 2314 disk storage facility
Journal of the ACM Vol 15 pp 577–589 October 1968

4 J ABATE H DUBNER
Optimizing the performance of a drum-like storage
IEEE Trans on Computing Vol C-18 No 11 pp 992–997 November 1969

5 W P MARGOPOULOS R J WILLIAMS
Characteristic problems on teleprocessing system design
IBM Syst Journal Vol 5 No 3 pp 134–141 1966

Manipulation of data structures in a numerical analysis problem solving system—NAPSS

by LAWRENCE R. SYMES

Purdue University
Lafayette, Indiana

AIMS OF SYSTEM

During the past several years considerable effort has been expended designing and implementing systems which are intended to provide extended capabilities for persons with mathematical problems to solve. Some of them in addition to NAPSS are CULLER FRIED, KLERER MAY, MAP, RECKONER, AMTRAN, and POSE. These systems can be classified as problem solving systems for applied mathematics.

Before the advent of these systems the research scientist or engineer used a procedural language such as FORTRAN or ALGOL when he employed the computer to aid him in solving a problem. Both of these languages, although they resemble mathematical notation more closely than machine language, are somewhat artificial and contain many unnecessary, from the user's point of view, rules. The artificial appearance and the rules must be mastered before the language can be used. Therefore the scientist or engineer is diverted from his main purpose into becoming a programmer. Even after he has learned the language, its complexity increases the probability of error, and thus reduces his efficiency.

In addition to these difficulties, the user with a mathematical problem had to use program libraries in order to obtain routines for solving commonly occurring problems. These libraries frequently were inadequate and almost always confusing. The routines often were poorly documented and performed little or no monitoring of the accuracy of the results. Thus the user of such a library had to know enough numerical analysis to select the best method for solving problems and to determine the accuracy of the results.

NAPSS has been designed to remove some or all of these problems and to offer several other desirable features. It, in some sense, endeavors to have man do what he is best equipped to do and to have the computer do what it is best equipped to do.

Six general techniques have been utilized to assist the user in stating and solving his problem.

First, the source language used to present a problem to the computer is similar to normal "text book" mathematical notation. This permits one to use the system without first having to intensively study the input language. It also reduced the probability of user programming errors, because the user is familiar with the notation.

Second, clerical statements used for dimensioning arrays and declaring variables are removed from the source language. These are tasks which the computer can easily perform but which are a constant source of errors if the user does them.

Third, NAPSS permits the direct manipulation of quantities other than scalars. These include numeric arrays, symbolic expressions, functions, and arrays of functions. This further allows the source language to resemble more closely "text book" form, and thereby leads to fewer statements; hence fewer opportunities for programming errors.

Fourth, solve statements are included in the source language. These statements permit the user to state a problem he wishes to solve in a concise, natural form. The user may include parameters such as initial values, the accuracy desired, the method he would like used, or he may omit any or all of the additional parameters. The solve statements invoke routines, polyalgorithms,[4,5] from a built-in library. They attempt to solve the user's problem automatically. They request additional information as needed and monitor the accuracy of the results in order to insure that it remains within the specified limits. The inclusion of these solve statements greatly reduces the burden normally imposed on the user. To solve commonly occurring problems with the aid of the solve statements, the user is only required to know how to define the equations for the problem; he is not required to know the numerical analysis involved or

even the method used. The method is selected by the system and the accuracy of the results is assured.

Fifth, on-line communication between the system and the user is provided by either a teletype or graphic display device. The use of these terminals bring the computer and the user closer together and consequently improve the user's efficiency.

Sixth, incremental execution of a program is allowed. This, combined with the use of on-line terminals, creates a closed loop between the user and the system. The user is able to monitor his program during execution and the system is able to request information from the user and point out errors when they arise. This eliminates much of the time that is wasted in preparing and submitting runs of a program which are unproductive because the user tried several fruitless cases, has an incorrect program, or has forgotten to initialize a variable.

NAPSS LANGUAGE

Rather than present a detailed description of the NAPSS language,[10,12] we describe a sampling of the allowable assignment statements.

The arithmetic expression in NAPSS permits the direct manipulation of numeric scalars, vectors, arrays, symbolic functions and variables which denote symbolic expressions. The user need not worry about the type or mode of the operands; rather, all that need concern him is whether or not the arithmetic expression is mathematically correct.

Several examples of arithmetic expressions and assignment statements appear below:

i) $D \leftarrow (B + C) \mid D * E \mid$

ii) ARRAY $\leftarrow$ ([3, 0:2] 1, 2, . . . , 9)/10

iii) E = V1 + V2 $\uparrow$ 2

iv) $F(X) \leftarrow A\,X \uparrow 2 + B\,X + C, (X < 0) \leftarrow A\,X \uparrow 2 - B\,X + C$

v) $G(X) = A\,X \uparrow 2 + B\,X + C, (X < 0) = A\,X \uparrow 2 - B\,X + C$

vi) $K(X)[1, 1] \leftarrow X \uparrow 2 - B, (X = C) \leftarrow -(X \uparrow 2) - B, (X > C)$

vii) $H(X, Y)[5, -2] = G'(X) + \int X \uparrow A\ (X \leftarrow 0 \text{ TO } Y)$

viii) $S \leftarrow$ "I AM A NAPPS STRING"

ix) $R[1, 1] \leftarrow S \mid \mid$ "ARRAY ELEMENT"

The left arrow operator ($\leftarrow$) indicates that the arithmetic expression on the right is to be evaluated and its value is to be assigned to the variable on the left. The value assigned to D is either a scalar or an array depending upon the operands in the expression on the right; while the value assigned to ARRAY is a 3 by 3 array.

The equals sign (=) has the more mathematical meaning. Statement three establishes that a future occurrence of E is equivalent to the expression V1 + V2 $\uparrow$ 2. Values are only substituted for the variables in the expression on the right of the = when a value of the variable on the left is needed. Thus if the value of V1 or V2 should change between the definition of E and the use of E this is reflected in the value of E. Variables defined to the left of an = are referred to as equals variables, and variables defined to the left of an $\leftarrow$ are called left arrow variables, or simply variables.

Statements four and five illustrate that a symbolic function may be assigned different definitions on different domains. The difference between statements four and five is similar to the difference between statements two and three. In the definition of F the variables A, B, and C have their current values substituted for them, while in the definition of G they do not. Values are only substituted for A, B, and C when a value of the function G is needed. Functions defined to the left of an = sign are called equals functions and functions defined to the left of an $\leftarrow$ are called left arrow functions.

Statements six and seven illustrate how arrays of functions are defined. All the elements in array of functions must have the same number of arguments and they all must be either left arrow or equals functions.

Statement eight assigns to S a string, and statement nine assigns a string to an element of an array.

Although NAPSS is intended primarily as a problem statement language, the features of a procedural language have been included to increase its power for the user who wishes to create a personal library of NAPSS routines. External and internal procedures may be written in NAPSS. The use of these facilities is optional. The casual user need not be concerned with the rules that procedures introduce, for he can employ the system on what is called console level.

On console level the user does not set up any procedures. Statements are entered without having to go through any initial set up, and are normally executed as they are received.

OVER-ALL STRUCTURE OF THE SYSTEM

The NAPSS system currently running on the Control Data 6500 at Purdue University consists of four main modules: the supervisor, the compiler, the interpreter and the editor. These modules are composed of 115

different routines, which are combined into 28 overlays. Almost all of the system is written in FORTRAN, with the exception of a few machine dependent operations which are restricted to "black-box" modules coded in assembly language. This is done to aid the goal of machine independence for the system.

The supervisor controls the flow into each of the three other modules. It distinguishes between NAPSS sources statements, which are processed by the compiler and edit statements, which are processed by the editor. The supervisor is also responsible for invoking the interpreter when a NAPSS statement is to be executed.

NAPSS source statements are transformed by the compiler into an internal text which the interpreter processes. This scheme was adopted for several reasons. First, the complexity of the elements to be manipulated and the absence of declarations require execution time decoding of operands. Second, it easily allows for extensions to the system. Third, it gives the user incremental execution. Fourth, it permits extensive error diagnostics and permits error corrections without having to recompile the whole program. Fifth, statements which are repeatedly executed are only translated once into internal text.

The internal and source text for each statement is stored in secondary storage. When a statement is to be executed, a copy of the internal text is passed to the interpreter. This reduces considerably the core storage required for a user's programme. Since the system is intended for use in an incrementally executing mode, no reference to secondary storage is normally required to obtain the internal text of a statement.

The system operates in one of two modes: suppress mode or execute mode. In the suppress mode, each statement is compiled into internal text and the internal and source text is saved on secondary storage for later execution. Suppress mode is entered by typing the statement .SUPPRESS. A block of statements which have been compiled in suppress mode may be executed at any time by typing the statement .GO.

The normal mode of execution is execute mode. Here, each statement is executed immediately after it has been compiled and a copy of its internal and source text saved in secondary storage. The system automatically enters suppress mode when the user starts a compound statement (a FOR statement) or a procedure. This is necessary because a compound statement cannot be executed until the whole statement is received and a procedure is only executed when invoked. The system re-enters execute mode automatically as soon as the compound statement or procedure is completed.

The memory of a NAPSS program is made up of a few pages of real memory which reside in core and a larger number of virtual pages of virtual memory which reside in secondary storage and are brought in and out of real memory. Two vectors (one dealing with virtual and the other with real memory) and several pointers are used to keep track of real and virtual memory.

Each element in the virtual memory vector is subdivided into three twenty-bit bytes. The first byte contains a flag indicating what type of information is stored in the page. The second byte is a switch, used when a page is in real memory to indicate whether or not a copy of the page also resides in secondary storage. The third byte contains the real page number the virtual page is in, when it is in real memory.

The elements of the virtual memory vector which denote available pages are linked together. Initially, the element for virtual page one points to the element for virtual page two and the last element contains a zero. When a page of virtual memory is returned to the system its element is again linked to the top of the list of available virtual pages.

The real memory vector elements contain one entry per real page. This entry is the number of the virtual page occupying it (zero of it is free). This pointer from real memory to virtual memory is used when a new virtual page is placed in real memory. The virtual page currently in the real page must be copied out into secondary storage if a copy of it is not already there.

The amount of core assigned to real memory is dynamic. Pages are removed from the top and bottom of real memory in order to obtain contiguous blocks of storage. Pages are removed from the top of real memory for two purposes: first, to expand the name table, and second, to obtain space for the work pool. Pages are removed from the bottom of real memory to obtain space for local name control blocks during the evaluation of left arrow functions. See Figure 1.

The work pool is used to hold arrays when performing array arithmetic. Requests for work pool space are always made in terms of words. However, the amount of real memory assigned to the work pool is always an integral number of pages. When a request is made for work pool space and the work pool is empty, the space supplied is zeroed. When space is requested for the work pool and the work pool is not empty, one of two situations arises. First, the space requested is less than the current size of the work pool. If the difference between the space requested and the current size of the work pool amounts to one or more pages, a corresponding number of pages is returned to real memory from the bottom of the work pool. Second, the space requested exceeds the current size of the work pool. If additional pages are obtained from real memory to satisfy the request, they are zeroed.

Virtual pages are assigned to real pages sequentially. Thus a virtual page is not removed until all real pages

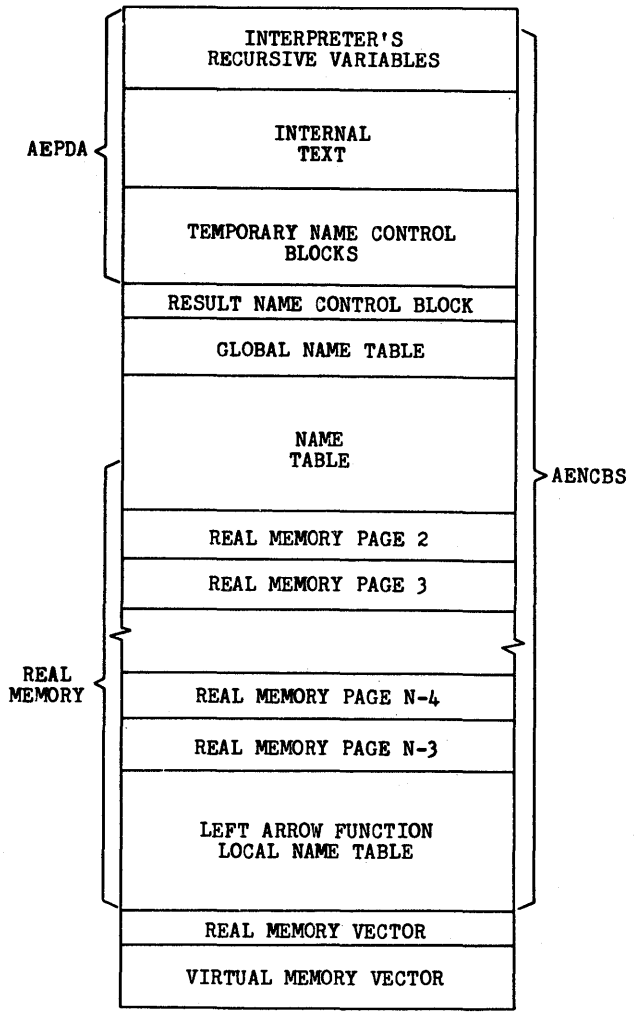


Figure 1—NAPSS memory organization

are assigned a virtual page. This sequential process may be broken whenever space is assigned to the work pool or to hold the local name control blocks for a left arrow function, since, after the space request is satisfied, the next real page to receive a virtual page may no longer belong to real memory. When this occurs the pointer to the next real page to receive a virtual page is reset to the first page now in real memory.

The algorithm for bringing virtual pages into real memory is further modified when the work pool returns a page to real memory. Since the page returned is empty, a virtual page may be placed in it directly, avoiding the possibility of having to save the virtual page currently there in secondary storage. Thus the normal sequential process is interrupted until all pages returned to real memory by the work pool are reused.

The system does not assign all of real memory to either the work pool or to space for left arrow function's local name control blocks. A request for real memory space is honored as long as two pages remain in real memory after the request is satisfied. If more space is requested than can be supplied, the request is modified to correspond to the maximum amount of space available. This permits the system to continue if this is adequate.

Two pages are required in real memory to facilitate the linking of virtual pages. With two pages in real memory the above algorithm guarantees that the previous and current virtual pages referenced remain in real memory. Thus they may be linked together if necessary, without having to save pointers and reread a virtual page to fill in link information.

Associated with each procedure is a name table containing entries for each variable, label and constant in that procedure. The entries, called name control blocks, are created during compilation when the name or constant is introduced. At this time it contains the name of the variable, and some basic attributes describing how the variable appears in the program. During execution the name control block is used to hold values, pointers to values and a complete set of attributes for the variable.

This double usage of the name control block entries poses no problem if compilation and execution are performed separately. But in NAPSS the normal mode of operation is to execute each statement as soon as it is compiled. Thus, three situations are possible when a variable is entered in the name table. First, the variable may never have been used before in the program. Second, the variable may have appeared before in the program but have no value assigned to it. Therefore, it is just as it was when the compiler last saw it. Here a limited compatability check is made between the two uses of the variable in the program. For example, the use of a name as a label and as a variable in an arithmetic expression is illegal. Third, the variable has appeared before in the program and has been assigned a value and a complete set of attributes. This enables more checking to be performed. However, the name table routine must not disrupt any of the attribute flags, for if any of them are changed the attribute may no longer correspond to the value associated with the name control block.

The name table is constructed sequentially. This method requires a minimum amount of space, and permits the name table to grow dynamically. The name table is expanded by removing pages permanently from real memory. This method of name table construction does require that the name table be searched sequentially. The search goes through the name table from bottom to top. This is done because frequently the greatest percentage of references to a variable occur in the immediate vicinity of its definition.

A variable which is declared to be global in N different procedures has $N + 1$ name control blocks associated with it. There is a name control block for the variable in the name table of each of the procedures in which it appears. Only compile time information and a pointer to the $N + 1$st copy is contained in these name control blocks. The $N + 1$st copy is in the global variable name table and contains a complete set of attributes for the variable and its value or a pointer to its value.

The $N + 1$st copy of a global variable's name control block is placed in the global name table when the first procedure is invoked in which the global variable appears, or when the variable is declared global on the console level (the portion of the program not contained in a procedure). When a global variable is added to the global name table and it already appears there, a check is made on the compatability of the attributes. An error results when they conflict. Otherwise a pointer to the $N + 1$st copy is placed in the procedure's copy of the variable's name control block.

A count is kept in the global name control block of the number of procedures referencing the global variable. When a global variable is no longer referenced, then its name control block is removed from the global name table and the storage associated with it is returned to the system.

A procedure is compiled when it is defined. To permit it to be linked into the program, the text generated uses only relative pointers to name table entries, and all linking between entries in a procedure's name table is done with relative pointers. This allows procedure A, for example, to be compiled as an external procedure and to be invoked either directly from the console level or from another procedure which itself is invoked from the console level. The name table for procedure A is placed in the name table after the last entry presently there when it is invoked and a base address is set up.

Variables which are not declared to be either local or global in an internal procedure are assumed to be known in the containing block.* After the procedure is compiled and a copy of its name table saved, a pass is made through the procedure's name table. This pass goes through the name table from top to bottom and places a copy of the name control block for each variable not declared to be either local or global, in the name table of the containing block. If the variable has appeared in the containing block, a compatability check is made between the attributes.

During execution only one name control block is used for the value and attributes of a variable which is not declared to be local or global. This is the name control block entry in the outermost block. The name control block in the internal procedure is linked to this when the interval procedure is invoked. The linkage is constructed so that only one step is required to obtain the value of the variable regardless of the depth of the procedure.

There are three types of name control blocks in different memory areas: ordinary, local for left arrow functions, and temporary. See Figure 1. Temporary name control blocks are used to hold temporary results during the evaluation of an arithmetic expression.

A central routine is used to decode variable name control blocks during execution. This routine determines the type of the name control block and handles the linkage between global, and non-local, non-global name control blocks. Three things are returned when a name control block is decoded: the attribute number, the data pointer field and the index of the array AENCBS of first word of the data pointer portion of the name control block. See Figures 1 and 2.

DATA STRUCTURES

A name control block is the basic unit of all data structures in the system. In some cases it holds the actual values of the variable, and in others it contains a pointer to the actual values and descriptive information. A name control block is made up of seven sixty-bit words of twenty-one twenty bit bytes. See Figure 2.

A name control block which denotes a numeric scalar contains the value of the scalar in its data portion. One or two words of the data portion are used depending upon whether the value is single precision real, double precision real or single precision complex.

When a name control block denotes a numeric array, the data portion of the name control block contains the actual bounds for the array, the declared bounds for the

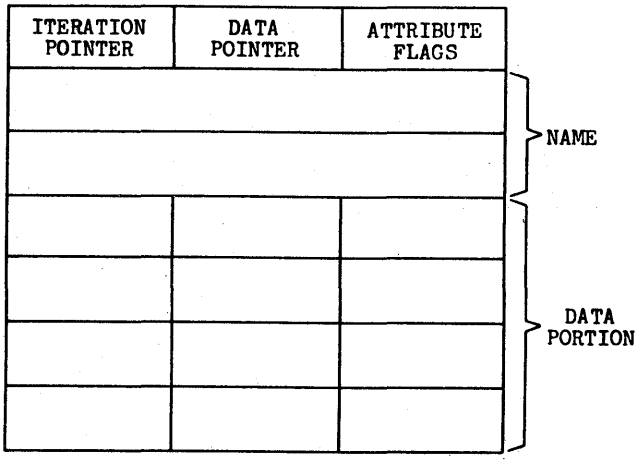


Figure 2—The layout of a name control block

* A block is either a procedure or the console level routine.

array (these may or may not have been specified by the programmer), and the number of dimensions in the array. The data pointer byte of the name control block points to where the actual array is stored, by rows, as a contiguous block. The array is stored as a contiguous block to speed up array operations.

If the data pointer byte of the name control block is non zero, a copy of the array exists in secondary storage in the array file. The data pointer is then the number of the record used to store the array and an index in the vector AEPAR.

The vector AEPAR contains additional information about the array. Each word in AEPAR is subdivided into three bytes. The first byte contains the reference count for the array. This is incremented by one each time the array appears in a left arrow function definition. The values of all non-parameter variables are fixed when a left arrow function is defined. The use of a reference count for arrays permits only one copy of the actual array to be kept, and if the non-parameter array variable is assigned a new value the value of the function will not change. The second byte contains the number of dimensions in the array. And the third byte contains the number of words in the array. The number of words is equal to the number of elements in the array times the number of words in each element. This factor is one for a single precision real array and two for a double precision real array or single precision complex array.

If the data pointer byte of the array's name control block is zero, the only copy of the array exists in the work pool, and the array is the result of the last array operation performed.

The work pool can contain anywhere from zero to three arrays. A counter is kept of the number of arrays in the work pool. In addition, for each array in the work pool the index of the first word of the array, the index of the first word of the data portion of the array's name control block, and the information contained in the array's AEPAR entry is kept. When an array operation is to be performed a check is made to see if any of the arrays involved already exist in the work pool. If they do, no reference to secondary storage needs to be made to obtain the operands. A check is also made to determine if the result of the previous array operation is an operand of the current array operation. If it is not the previous result array must be stored temporarily in secondary storage.

A name control block which denotes an equals variable contains the virtual page number of the first page used to store the internal text for the expression in its data pointer byte. The first word of each virtual page is used for linkage. The link contains the virtual page number of the next page used to hold the text of the expression or zero if the page is the last. When an equals variable is an operand of an arithmetic expression this internal text is evaluated to obtain a value for the equals variable.

If a name control block denotes a scalar symbolic left arrow function, the data pointer contains the page number of the first virtual page used to store the internal text of the arithmetic expression for the first domain of definition. The first byte of the fourth word of the data portion contains the number of arguments of the function.

The first four words of the first virtual page used to store the internal arithmetic expression text for each domain contains a set of pointers. The first word is used to link together the pages required to store the internal text for the arithmetic expression for the domain. It contains the virtual page number of the next virtual page used. A zero link denotes the last page. The next three words are subdivided into nine bytes. The first byte contains the number of words of internal text in the boolean expression for the domain. This is used when the boolean expression is being moved prior to its evaluation. The second byte contains the reference count for the function. If the function appears in the definition of another left arrow function this is increased by one so that only one copy of this function needs to be kept. The third byte is the virtual page number of the first page used to hold the text for the boolean expression for the domain. This byte is zero if the domain has no boolean expression. The fourth byte contains the number of virtual pages that are required to hold the local name table for the domain. The local name table contains a name control block for each non-parameter variable appearing in the boolean and arithmetic expression for the domain. This is necessary so that the value of these variables can be fixed when the function is defined. Byte five is unused. Byte six contains the virtual page number of the first page used to hold the local name table for the domain. Byte seven contains the number of words of internal text in the arithmetic expression for the domain. Byte eight is unused and byte nine contains the virtual page number of the first page of internal, arithmetic expression, text for the next domain. If this byte is zero, there is not another domain defined for the function.

The virtual pages used to store the text for a boolean expression or a local name table are linked together by the first word of each page. A zero link specifies the last page.

The name control block for a scalar symbolic equals function contains the same information as a scalar symbolic left arrow function. The text for the function is also stored in a similar fashion except that in the first virtual page used to store the internal text for a domain's arithmetic expression bytes two, four and six are not

used. There is no local name table required for an equals function since all non-parameter variables appearing in the function definition assume their current values when the function is evaluated. There is no reference count because if an equals function appears in the definition of a left arrow function, a copy of the equals function must be created. While the copy is being made the equals function is transformed into a left arrow function to insure that the values of all non-parameter variables are fixed.

If a name control block denotes an array of symbolic functions, it contains the same information as a numeric array name control block. In addition the first byte of the fourth word of the data portion contains the number of arguments in each of the functions.

The array is treated as if it is an array of real single precision numbers. Each element contains the virtual page number of the first page used to store the arithmetic expression text for the first domain of the element's definition. If an element is not defined, its value is zero. The text for the definition of each element is linked together in the same manner as a scalar symbolic function.

NAPSS is not designed for string processing but it does allow the user to create strings, concatenate them and assign them to variables. This is done to permit the programmer to label his output. The data pointer byte of a string valued variable's name control block contains the number of the string. The string number is the index of an entry in the string relocation vector. Each entry is subdivided into three bytes. Byte one contains the index of the start of the actual string description in the string picture table. The second byte contains the reference count for the string. The reference count designates the number of times the string variable has been concatenated to form another string, plus one. The third byte contains the index of the first word of the data portion of the name control block for the string variable.

The string picture table contains a description of each string. Several entries compose the description of a string. Each entry denotes either a literal string, a reference to a previously defined string variable, or the end of a string picture. An entry in the string picture table is subdivided into three bytes.

If byte one is not zero the entry describes a literal. Byte one is the number of characters in the literal, byte three is the number of the virtual page in which the literal is stored, and byte two is the displacement on that page to where the literal begins.

Each word in a virtual page used to hold string literals is subdivided into three bytes. A literal is divided into segments of three characters. Each segment is stored in a byte. If a string literal will not fit in the current string page, the literal is broken. As many segments of the literal as possible are placed in the current page and the remainder are placed in a new string page. When this occurs two entries are placed in the string picture table. This avoids the problem of linking pages used to hold string literals. The maximum length of one string literal is 576 characters.

If byte one of a string picture table entry is 1313, then the entry denotes the null string. It has no length and does not require any storage, so byte two and three are unused.

If byte one is zero and byte three is not 501, the entry denotes a reference to a previously defined string variable. So that a new copy of the previously defined variable's string is not created, byte three contains the index of its entry in the string relocation table. When this occurs the reference count in the relocation table for the variable is increased by one.

If byte one is zero and byte three is 501, the entry denotes the end of a string picture.

When a name control block denotes an array of strings, it contains the same information as a numeric array. The array is treated as a single precision real array. The elements of the array contain the indices of the entries in the string relocation table for the string descriptions. If an element is undefined, its value is zero.

As can be seen from the descriptions of the various data structures, the primary concerns in their design has been to facilitate their use as operands while at the same time reducing the amount of physical storage required.

ACKNOWLEDGMENT

The work was supported in part by NSF Contract GP–05850.

REFERENCES

1 G J CULLER
Mathematical laboratories: A new power for the physical sciences
Interactive Systems for Experimental Applied Mathematics Klerer and Reinfelds eds pp 355–384 1968

2 M KLERER J MAY
An experiment in a user-oriented computer system
Communications of the ACM Vol 7 No 5 pp 290–294 1964

3 M KLERER J MAY
A user oriented programming language
Computer Journal Vol 8 No 2 pp 103–109 1965

4 J R RICE
On the construction of polyalgorithms for automatic numerical analysis
Interactive Systems for Experimental Applied Mathematics Klerer and Reinfelds eds pp 301–313 1968

5 J R RICE
A polyalgorithm for the automatic solution of non-linear

equations
Proc of ACM 24th National Conference pp 179–183 ACM Publication P–69 1969

6 A RULYE J W BRACKETT R KAPLOW
The status of systems for on-line mathematical assistance
Proceedings ACM National Meeting pp 151–168 1967

7 S SCHLESINGER L SASHKIN
POSE—A language for posing problems to the computer
Communications of the ACM Vol 10 No 5 1967

8 R N SEITZ L H WOOD C ELY
AMTRAN—Automatic mathematical translation
Interactive Systems for Experimental Applied Mathematics Klerer and Reinfelds eds pp 44–66 1968

9 A N STOWE R A WEISEN D B YNTEMA J W FORGIE
The Lincoln reckoner: An operation-oriented on-line facility with distributed control
Proceedings FJCC p 433–444 1966

10 L R SYMES R V ROMAN
Structure of a language for a numerical analysis problem solving system
Interactive Systems for Experimental Applied Mathematics Klerer and Reinfelds eds pp 167–177 1968

11 L R SYMES
Evaluation of NAPSS expressions involving polyalgorithms, functions, recursion and untyped variables
Purdue University Technical Report CDS TR 33 1967

12 L R SYMES R V ROMAN
Syntatic and semantic description of the numerical analysis programming language (NAPSS)
Purdue University Technical Report CSD TR 11 Revised 1969

A study of user-microprogrammable computers

by C. V. RAMAMOORTHY and M. TSUCHIYA

The University of Texas
Austin, Texas

PART ONE—USER-MICROPROGRAMMABLE COMPUTER

The user microprogrammable computer as the fourth generation computer is investigated from the user's point of view. In the first part of the paper microprogramming and its concept as well as the problems and requirements incurring its use in various applications are discussed. The current status of the microprogrammed computer is also studied to indicate the differences of philosophy in microprogramming. A number of suggestions are made for the design of fourth generation user-microprogrammable computers.

An algorithm for determining the optimum size of the microprogram store is presented in the second part of the paper.

INTRODUCTION

In many ways the evolution of computer architecture can be likened to that of the human species on this earth. Possibly the current computers represent those fossilized representatives of the prehistoric times with enormous bulk and little ability for adaptation to the environment. But nature's evolutionary processes always endeavored to adapt the species to their environment. In a similar sense evolution of the computer architecture had been towards its adaptation to its use, i.e., to the problems the computers are intended to solve.

In a sense, the different generations in the architecture of computers can be distinguished as follows. The separation of procedure and data as exemplified by Babbage and Aiken represented the first generation. Their integration in one storage device and the classification and consolidation of different functions into distinct functional units signaled the second or von Neumann generation of computers. Typically the latter exemplified the permanent irrevocable wiring in all functional tasks (machine language instructions). The third generation which saw the emergence of multiprogramming and time-sharing made the control logic more flexible by storing the sequences of elementary functional operations in a read-only memory (ROM) called a microprogrammed storage or control storage. Primarily this simplified the engineering design of the control unit, the testing and the maintenance functions, and provided the advantage of having a large instruction repertoire which was required for reasons of compatibility in a family of computers, e.g., IBM 360. The fourth generation yet to come has opened up another dimension, that is, adapting a computer to the problem environment efficiently. One of the basic techniques of achieving this is via user generated and user alterable microprogramming techniques.

In this paper we shall take a cursory look at the problems and prospects of new generation of machines which could provide dynamic adaptation to the user's instantaneous needs, particularly when the resources in the system are shared by a multiplicity of users. Since the day of massive computer utilities is not far off,[8] the type of computer we are considering here could be a basic building block of the computer utility of the future. It is in this context we shall examine what we believe the first step towards problem adaptable computers, namely the user microprogrammable computers.

Non-microprogrammed and microprogrammed computers

Wilkes[15,16,17,18] the father of microprogramming, suggested microprogramming as an orderly way of designing control sequences to execute machine instructions which used many common programming techniques to advantage, such as program branching, and the sharing of common sequences amongst machine instructions (subroutine concept) to provide tremendous flexibility. In all computers programmed instructions

reside in a memory which can be altered. In non-microprogrammed machines the control unit which sequences all the instructions is implemented with hard-wired components. Microprogrammed computer incorporates stored program rather than hard-wired implementation of the control unit. Thus the control unit's operating characteristics (architecture) can be changed without changing the physical implementation of the hardware. Whereas a fixed control machine can be efficient to one class of applications but less efficient to others, the microprogrammable machines can be adapted easily to different problem-oriented environments.

BASIC CONCEPTS

A computer generally translates a sequence of programmed orders into a sequence of machine codes which is the only form that the machine can recognize. The computer then interprets and executes the sequence of machine codes. When a machine code operation is performed, transfers of information occur among the functional components (e.g., registers, memory, adder, etc.) of the computer. The communication between functional components in turn is controlled by a set of primitive machine operations which consists of opening and closing gates and circuits between registers and basic logic elements within the control store of the computer.

Conventional fixed control computers

In these, the machine code is interpreted and executed by a completely wired-in set of circuits in the control unit of the computer. Thus the computer executes the particular, but small, set of machine codes (i.e., machine language) efficiently. It accepts only one machine language, however, and, as discussed later, it is inflexible in terms of its applicability. As the vocabulary of the language increases, the hardware complexity also increases.

Any elemental operation such as a register-to-register transfer of information performed during the execution of a machine instruction is called a *micro-operation*. A *micro-instruction* specifies one or more micro-operations that could be performed in a fixed time interval. The micro-instruction has predetermined formats which specify internal data flows. Generally it is stored in one or more locations in a fast memory called a *micro-program memory* (sometimes also called a *control store*). A *micro-program* is a set of micro-operations used to effect a single machine code of user machine. Every machine code then is considered to be programmed by the proper arrangement of micro-operations similar to the concept of machine code programming. In this context, if microprograms are stored in the modifiable (writable) memory rather than hard-wired, the machine codes may be altered and redefined by changing the arrangement of micro-instructions to suit the particular needs.

Microprogrammable computer

The microprogrammable computer is a multi-lingual, multi-purpose, flexible computer. Its machine instruction is performed by a microprogrammed stored subroutine. Although there may be a loss of computing speed when stored micro-instructions are fetched from a memory, it is offset by the use of the high-speed memory and the simplicity of the hardware construction.

The most significant advantage of the microprogrammable computer is its flexibility offered to the user. It is a multipurpose (note the difference from "general purpose") computer and is flexible enough to be particularized according to the user's application environment. It is then the user's responsibility to take full advantage of the microprogrammable computer.

The level of control

There are typically two types of micro-instruction formats which characterize the level of control exercised on the elementary operations. In one type (the function/field type), the fields of the micro-instruction control the gates on the individual data paths of certain elemental functions (Figure 1). Each field of the instruction constitutes a micro-operation. The micro-instruction thus controls data flows within the machine *at the lowest level* (e.g., IBM 360/50). The more complex the machine structure becomes, however, the larger becomes the number of fields in the micro-instruction. Although microprogramming in this instruction format can be tedious because of the fineness of control, the specification of a macro-instruction can be

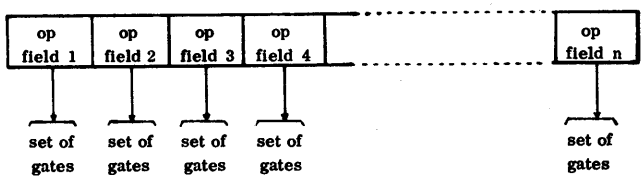


Figure 1—Function/field type micro-instruction format

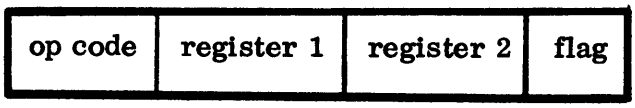


Figure 2—Machine-code type micro-instruction format

compact and efficient. A multiplicity of fields in a micro-instruction which is executed in one machine cycle permits the microprogrammer to specify and use fully the parallel processable opportunities available in the system control. This type of microprogram control is useful particularly in specifying the operating system functions of a large computer because (1) efficient and compact implementation of highly active operating system functions can reduce in time the wasteful oyerhead and (2) such functions once implemented become building blocks for others.

In the other type (the machine code type), the micro-instructions, or for our discussion, "mini"-instructions are sequences of commonly occurring elementary functions (Figure 2). The format of a mini-instruction is similar to that of conventional machine code instruction: one operation code field, one or more operand fields, and, optionally, a flag field for special conditions. Operands in mini-instructions are, in general, originating and/or destination registers of some information to be transfered. A mini-instruction may be characterized as follows: each mini-instruction represents an elementary functional task, and any macro-instruction (i.e., machine code) can be built up from mini-instructions. A mini-operation differs from a micro-operation in that the former represents a sequence of gating operations requiring a number of basic clock cycles while the latter requires one clock cycle. The mini-operation may be considered as consisting of a few micro-operations. In fact many mini-instructions resemble basic machine instructions in some simple computers such as an IBM 650. The "ADD" mini-instruction of the Interdata Model 4 computer, for example, is performed in three micro-steps as follows:[1]

(1) transfer to the adder the contents of the *A* register;
(2) transfer to the adder the contents of the register specified by the Source (*S*) field of the instruction;
(3) transfer the contents of the adder (i.e., sum) to the register specified by the Destination (*D*) field of the instruction.

In the instruction, the registers are designated by alphanumeric names which, in turn, are translated into and represented by the four-bit *S*- and *D*-fields (Figure 3). Mini-operations are either wired in the machine or activate micro-programs in the control memory. In the case of the latter, the instruction execution time is longer simply because of additional memory references. Although mini-instructions are not suitable for specifying overlapping operations, data flows within an individual mini-instruction may be performed in parallel. The mini-programs are generally user-oriented and easier to write and debug. As in the case of high level languages, the storage requirement for mini-programs are smaller than their microprogram counterparts (up to 20 per cent) but additional control memory or wired-in logic is necessary for their interpretation.

Users

The motives for the user to use microprogramming are many. In order to discuss problems from the user's viewpoint, we classify computer users largely into two categories according to their main motives and interests in using microprograms. The first, called the *owner-user*, consists of those who own the computer system, maintain, manage and schedule its use. The administrator of a computer center, systems programmer, etc. are in this category. They do not necessarily design or run application programs. They are mainly interested in the maximum utilization of the computer resources by the customer programs.

The second is called the *customer-user* who actually uses the computer to run his programs. Any one individual could be both owner-user and application-user depending on his interest and the type of job he runs at the moment. This includes those who use the system to perform computational tasks not related to the management and upkeep of the computer system. The customer-user is mainly interested in the maximum convenience and dependability such as better turn around time or response time etc., for processing his jobs.

The difference of purpose and interest in computer usage necessarily differentiates the users' interests in microprogramming.

MICROPROGRAMMING SITUATIONS

The two types of users try to restructure the computing system in two different ways; the owner-user tries to maximize the utilization of his equipment by the customers and the customer-user tries to maximize the performance of the equipment in solving his problem.

Microprogramming for owner-user

The primary motive for the user-owner wanting access to microprogram store for its content modification is to accommodate system expansion or change.

Emulation

One primary area is emulation which reduces the reprogramming costs due to change over to new systems having many similar instructions and addressing structures.[13] *Emulation* can be loosely defined as a combined software/hardware interpretation of the machine instruction of one machine by another. In a conventional non-microprogrammed machine, the emulation to be successful requires that the instruction formats and repertoire must be very similar. A microprogrammed machine, on the other hand, can emulate a wider variety of machines. In particular however, experience indicates that word lengths of the host computer must be same or some convenient multiple of the target computer being emulated. It has been found desirable that the number of arithmetic registers in the host machine must at least be equal to or greater than that of the target machine simulated for efficient emulation. Another desirable feature to have in the host microprogrammable machine is that there be a number of toggles (flip flops) which can be used for setting and branching by appropriate conditional micro-operations.

Expansion of I-O and Memory

Possibly the greatest benefit to the owner-user is accrued when microprogramming is used to accommodate the addition of new I–O devices, or systems adjuncts like associative memories or additional memory capacity. One can categorize the added chores that the system has to accommodate due to the expansion of I–O and memory. In the case of added I–O, they can be increased interrupt activities, special I–O formating, manipulation of variable length fields, new I–O commands (if new devices are added), and special error recovery procedures.

In the case of increased-memory capacity, there is the problem of addressing associated with the increase in the number of pages in the paging schemes. New microprograms can be written to reinterpret old I–O commands to take into account the change.

Adding New Processing Versatility to System

The owner-user to accommodate scientific customers may provide specialized array of new commands or fast subroutines (which can be partly microprogrammed) not provided originally in the system architecture of the computer, e.g., commands for multiprecision, floating-point arithmetic, radex conversion, etc.

Redistribution of Operating System Functions

Occasionally a redistribution of system functions between hardware, software and firmware may be in order. Principal reasons may be new applications and new devices added into the system. For example, the introduction of a real time control system may require periodic monitoring and instant servicing of high priority interrupts. Frequent use of a function or the immediacy of response can be good reasons for microprogramming part or whole of the function, rather than leaving it in the software.

System reconfiguration due to an application need or failure of some system device is another instance where redistribution of operating system functions may be desirable. Incurring changes can also be met by modifying microprograms of some functions.

Microprogramming for customer-user

To the customer user, access to microprogram control has many advantages which are fraught with headaches to owner-user and the system. We shall briefly outline a few avenues of benefit to the customer-user.

Real time environment

Microprogramming may be a solution when machine instructions and assembly language programming can not keep up with process, where the process is primarily CPU limited.

Application orientation

Where it is desirable to execute the problem in a manner natural to the process such as in certain problem oriented machine languages,[6] it becomes desirable to interpret application languages and data structures in their natural form to facilitate on-line debugging, modifying and program building. In these cases, the statements in application language may be executed directly via microprogrammed interpretation. To carry this further, by having access to a dynamically alterable microprogram store, one can particularize the computer dynamically to match with the varying problem types and environments as encountered, for example, in a jobshop environment. In this manner, the process-

oriented, higher-level language can be executed easily via a microprogram interpreter.

Performance enhancement of production Programs

When some procedures of a production-type program which is run frequently on different sets of data, e.g., payroll, are executed time and again, they can be microprogrammed to enhance the performance of the program and reduce the overall execution time. Statistics gathering for the usage frequency and execution time of procedures can be performed concurrently with execution processes again by the use of microprogrammed adjuncts. Thus by analyzing the usage statistics it is possible to microprogram selected parts of the program as well as selectively storing machine language instruction strings in high speed control memories to improve the execution time of the program. In the second part of this paper, we shall develop an algorithm for this purpose.

ADJUNCTS FOR USER APPLICATIONS

We shall review some key user applications and consider what other hardware–software aids needed in addition to dynamically alterable microprogram storage.

Realtime environment

In the area of real-time control applications where vigilance to high priority interrupts and immediacy of response are essential, the detection and classification of interrupts is conveniently performed by a fast associative memory. An associative memory is predicated here since it provides the flexibility of assigning any priority to the interrupt lines and having so assigned, the identification and servicing can proceed accordingly. Real time applications also need an access to clock (timer) for purposes of sampling, etc. Amongst the software adjuncts there is a need for a local monitor (executive) to provide "a better collaboration" and communication between the real time process and the system executive. Interrupt monitoring and control of associative memory could be more beneficially microprogrammed.

Simulation

In the area of simulation of dynamic processes, frequently used subroutines such as function generators, random number generators and transcendental function generators are best microprogrammed. Real time clocks and associative memories form useful hardware adjuncts in this area also.

Pattern recognition

The area of pattern/symbol recognition has developed many picture processing languages. Many of them require extensive matching and bit manipulations. In particular, they involve neighborhood processing of clusters and adjacent bits. Here again, microprogramming with associative processing could ease the task.

Alterable microprogram store

Best benefits of microprogramming are accrued when the user has access to certain specified portions of microprogram store and has ability to modify its contents. The system can use a relatively fast possibly unalterable read-only memory to interpret basic machine operation codes and certain frequently used subroutines like radex converters, etc. The dynamically alterable microprogrammed memory can be shared between the system (owner–user) and the customer–user. Most preferably the customer user portion must be paged. For most common functions, the system with a modest instruction repertoire would need about in the order of 10^6 bits or $2K$ words of approximately 48 to 64 bits per word. The access time of the microprogram memory must be at least between $\frac{1}{3}$ and $\frac{1}{5}$ times that of the main memory. The user microprogram store must be paged or otherwise it may impose costly allocation and garbage collection problems with multiple concurrent users. The cost of renting of the microprogram memory must be reasonable to the customer user. Of course, because of its very nature, the user must be knowledgable at least to the extent of using it to better cost performance ratio. It is obvious that its use will be favored by production-type processor-oriented problems rather than otherwise. Also, the system owner must allow larger quantum of computational time to the users using microprogramming for the very reason that since their memories are expensive their cost advantage lies in their frequent usage.

Micro-instruction structure

To be readily usable, the micro-instruction structure must be simple. Some third generation machines such as an IBM 360/50 require so much knowledge of the

sophisticated, internal organization of the machine control that micro-programming for them is a nightmare. It is essential for easier microprogramming that the microprocessor of the "visible" machine which the microprogrammer sees and uses be simple.

This also brings up a need for an assembly language or a higher level language for microcode. The customer-user, unlike the owner-user, would use the microcode only if it is simple and convenient to write and debug. Micro-instructions must be syntactically sound and should not contain any "unnatural" funnies. Assemblers and compilers, on the other hand, tend to reduce the "tightness" or compactness of the final microprograms. Thus, there is a possibility of writing microprograms at two levels: micro-level and mini-level. The *micro-level operations* are the most elemental operations beyond which the machine cannot be controlled. They may represent basic gate level controls. Microprogramming is the method of describing these micro-level operations to execute certain computational tasks. This mode is beneficial to system owner–user in implementing I–O controls, and certain frequent operating system functions. The key issue is that efficient code is essential here and programs once written become building blocks for others. The *mini-operations* are sequences of commonly occurring micro-operations which are characterized by the following: (a) Each mini-operation represents an elemental functional task that the system can perform, and (b) any general purpose instruction repertoire can be built up from combinations of mini-operations. Specifically, the mini-operations will be user-oriented. The motive for introducing mini-operations is twofold. Mini-programs are easier to write than microprograms and hence easier to debug. Just as in the case of higher level language, the amount of storage space needed to store mini-programs will be vastly smaller than that required by microprograms as much as 20 per cent. Of course, the execution time for mini-programs will be longer than execution of microprograms due to an additional level of interpretation. Mini-level operations could provide parallel processing capabilities whereas microprograms can only be sequential. One version of mini-instructions can be subsets of Iverson's *APL* operators which make array representation and parallel manipulation possible. This again provides economy in notation and storage.

It appears that re-entrant types of microprograms with physically separate high speed scratch pad storage area may be advantageous in the future. To summarize, the goodies required for multi-user microprogram sharing and use are similar to those in a conventional multi-programmed area.

Another requirement would be the addressing of arrays and instructional information stored in the microprogram memory. Since it is believed 80 per cent of all numerical calculations involve matrix manipulation such processor limited computations can benefit by matrix type addressing facilities and reduce the user storage requirements of microprogram memory. Since most of the addressing within a microprogram memory is always within the close proximity of the address of micro-instruction, method of "proximity" addressing (e.g., incremental addressing) can be helpful. Since logic is cheap, arithmetic transformation of addresses may be very helpful as against a random transformation as in paging.

Privilege among users

It is obvious that not all micro-operations must be available to the customer users whereas the system (owner–user) should have access to all. Thus the system operates in a privilege mode, having access to all vital controls in the systems. In the restricted privilege mode, the customer-user will not have access to:

(1) Any micro-operations dealing with address transformation, memory or I–O barricades;

(2) Any micro-operation dealing with hardware functions of the executive, like interrupt handling and polling mechanisms, etc.

Let us now consider the approaches for implementation of these. One obvious solution is to have two microprogram memories, one which can execute the unrestricted operations of the system and the other the restricted subset allowed to the user. The operation decoder will not execute any system micro-operations which come from user portions of a program memory.

Parallel surveillance and debugging

When multiple users are using the microprogrammed control memory concurrently, it is essential for the system to provide close surveillance. The possibility of one program clobbering another program unintentionally is high when the program being executed is new or being debugged. The tangible approaches to the problem include any or all of the following:

(1) Restricting the users to a subset (unprivileged) of micro-operations;

(2) Sequencing each machine instruction in two modes: normal mode and debugging mode. In the debugging mode each instruction will be scanned for possible syntax errors and conflicts in address space, etc. Also sentinels, breakpoints and test address information will be serviced. Any desired intermediate

computations will be saved. In the normal mode the surveillance operations will be suppressed.

In the second approach instead of interweaving the surveillance and execution sequences, one can have two distinct micro-program sequencing units, one unit for normal instruction execution and the other for overseeing and interrogating the execution sequence. This, of course, involves the availability of multiple microprogram control units. The use of a supervisory microprogram monitoring each machine code operation is equivalent to each program having its own individual supervisor.

Relocatability

Relocatability of a program implies that it can be put into any contiguous set of addresses in a memory and executed with minor reinterpretation of its address fields. Specifically, it implies that the addressing is relative to a reference. By having relocatable features, the micro or mini-programs can be stored contiguously in the microprogram memory and hence it is possible to serve a number of users of microprograms by swapping them in and out of the expensive microprogram-storage.

PERIPHERAL AREA

The first good use of microprograms came in the peripheral area when IBM developed the I/O channel for their pre-System 360 computers. The channel is a small microprogrammed computer itself; it communicates with the central processor and controls the various I/O devices. The channel transfers the required amount of data between locations in main memory and I/O devices, protects against unauthorized transfer of information into main memory, signals the processor units of I/O operation status by means of interuption and permits concurrent activities of the central processor and I/O devices (i.e., multiprogramming). Microprograms of the channels thus provide flexibility since they handle a wide variety of I/O devices as well as complex communications with memory and the processor.

Microprograms are also used in satellite computers of a large computer system. The satellite computer is mainly used to control the activity of various I/O devices. In this sense, it functions as a channel to the main computer. It can transfer information to and from the central computer, check word parity, and store information in the specified location of the specified storage device. Microprogramming is employed to enhance the efficiency and flexibility of satellite computers, and to control a variety of I/O devices.

FUNCTIONAL REQUIREMENTS

Micro-operations available to the users

All microprograms available to the system are classified into two categories: normal micro-operations and privileged micro-operations. The normal micro-operations are those that may be used by the customer–user in his microprogram. The privileged micro-operations are those that may be used only by the system-user.

Protection may be realized by using two microprogram memories: one for the customer-user and the other for the system-user. If they are used strictly by their designated users, encroachment is fairly easily prevented.

Visible machine

The visible machine, that is, the machine organization as seen by the microprogrammer must be simple. The internal structure of the machine should be well-organized so that the data flow among functional units may be seen easily via micro-operations.

Functional congruity

The incongruity among functional units requires much housekeeping operations and causes clogging of internal data flow and, therefore, must be minimized. Consider, for example, a functionally incongruous machine with 2-byte wide memory data path, 2-byte wide registers and 1-byte wide adder. It is apparent that for any simple 2-byte addition, a value has to be divided into high- and low-order parts in order to conform to the adder, then computation must be performed on each part sequencially. Every 2-byte addition requires two passes to the adder.

A microprogrammed computer must be designed such that a set of common microprogrammed sequences can be used in more than one way. To illustrate this view, consider now a computer which has an adder that can be set to add/subtract in either binary or in binary-coded-decimal in groups of 4 bits.

Then the following two algorithms will perform multiplication and division in either binary or decimal number systems depending on the setting of the adder. In other words, they have identical sequencing for decimal and binary multiply and divide procedures.

The multiplication algorithm which gives a two-word product when a word A is multiplied by a word B is as follows.

1. Store A.
2. Compute and store $4 \times A$.
3. Consider the lowest order four-bit group of B as being $4\alpha + \beta$ where α and β are each two bit numbers. Add $4A$ α-times and A β-times to form a partial product.
4. Shift both B and the partial product four places to the right.
5. Repeat steps 3 and 4 for the next higher order four-bit group of B. At the end of the procedure, a two-word product is obtained.

The following algorithm for division may be micro-programmed advantageously to divide a two-word number A by a one-word number B yielding a one-word quotient and a one-word remainder.

1. Store B, then compute and store $-4B$.
2. Add $-4B$ to the high order part of A α-times until the accumulator goes negative. α will be in the range of one to four.
3. Add B to the accumulator β-times until the accumulator goes positive. β will be in the range of one to four.
4. Record the first digit of quotient, the high order two bits being $\alpha - 1$ and the low order two being $4 - \beta$.
5. Shift the diminished A four places to the left and repeat the steps 2, 3 and 4 as many times as the number of 4-bit groups in one word. The digits yielded are the successively lower order digits of the quotient, and the number finally left in the accumulator is the remainder.

These processes are applicable when the negative quantities are represented by their two's (ten's) complements. The reader may verify that the algorithm is valid for both binary and *BCD* arithmetic. It is obvious that this is more economical in terms of microprogram memory space than having two separate microprograms: one for each binary and decimal operation. Microprograms arranged in this manner will not only economize the space and use of microprogram memory, but will simplify the computer organization.

Parallel surveillance on all user operations

In a multi-user, multi-processor environment, concurrent operations can encroach each others working resources. It will be necessary to barricade each user from innocent mistakes of others, particularly in their microprograms. The micro-programming sequences can be so designed that instrusions, and deadlocks could be prevented or if they occur, the damage to other programs is either recoverable or minimal.

Allotment of microprogram memory space among multiple users

Allotment of modifiable microprogram memory (MMM) space among multiple users is a problem of the operating system. The operating system assigns each job a priority for the use of MMM and when some MMM space becomes available it allots the space to the incoming jobs according to their priority.

Generally, microprograms that perform elementary operations can be shared by all user programs to prevent unnecessary duplication. Therefore, commonly and frequently used microsequences should be coded as macros and stored in a designated space where all the users have access. This would reduce the amount of MMM space allocated for the individual job and certain jobs should be processed within the scope of micro-programmed macros.

Addressing the microprogram memory

A user program may be coded into regular machine code or a set of microprograms or a combination of both. If it is coded into machine code, it is stored in the core memory store. If, on the other hand, it is coded into a set of microprograms, it should be brought into the control memory. When parts of a program are coded into machine code and others are micropro-grammed, communication linkage for control, i.e., a uniform addressing scheme between the two types of memory must be established. In order for control to be shifted between the two memories, the starting address of a microprogram and return address of the program in the core memory must be specified.

Protection within control memory

When a number of user microprograms reside in the modifiable microprogram memory there arises the possibility of innocent and unintentional encroachments into one another's microprogram space. A number of schemes prevalent in the area of current time sharing computer systems can be used to prevent unauthorized memory encroachments. We shall briefly list them as follows:

1. the procedures can be written in the "reentrant" code and the modifiable part of the microprogram would be located at a distinct location;
2. the microprogram memory space can be "paged" and access to each page would be checked for user identification and protection.

Simpler schemes described below can be also adequate.

1. Restrict the customer user's access to a certain area of the microprogram store. In this way, the vital functions implemented in the owner-user's microprogram store will be protected.
2. Process programs in two modes: debug mode and the normal mode. New programs are run in the debug mode first.

In the debug mode, the microprograms sequencing the user's program will check for violations of his address space, etc. Once fully debugged, the programs can be run in the normal mode and executed.

CONTEMPORARY MICROPROGRAMMED COMPUTERS

Currently, a large number of microprogrammed computers are available on production basis. They are not intended, however, to allow the user a flexible use of microprograms. In fact, with a few exceptions, their microprograms are unalterable. The VIC–I computer of RCA, for example, has an unalterable read-only memory for its microprograms. Its main design objective is high reliability for aerospace applications. Every macro-instruction (machine code) is performed by a set of basic micro-operations and is capable of being executed in a variety of ways through various combinations of microprograms. Microprograms are also effectively used to implement a provision for "graceful degradation" through error sensing circuits and automatic rerouting.

The Micro 800 is a small-scale, microprogrammed computer with a fast read-only memory of 220 nanosecond cycle time. In this computer, good uses of microprograms are observed. The macro-instruction is fully dependent on the microprograms and is electrically alterable within the capabilities of the hardware. Main memory word length (in multiples of 8, 9, or 10 bits) and I/O interrupt servicing are also controlled by microprograms. A special micro-memory board can be inserted to perform system diagnostics. The manufacturer provides a Micro 800 simulator written in Fortran IV for microprogram logic design and debugging on a variety of computer systems.

The IC–9000 of Standard Computer Corporation is a small-scale computer and so far the most versatile and powerful with respect to microprogramming capabilities. It is a relatively expensive micro-processor with many sophisticated features as well as a fast read-only microprogram memory (and fast-writable memory available at increased cost). Macro-instructions are microprogrammed with the exception of optional "Language Boards" which perform preliminary decoding of target language instruction generating a transfer address in the microprogram and setting various conditions. The IC–9000 as a versatile microprogrammed data-processor has a number of advanced capabilities such as micro-subroutine nesting, limited instruction overlap, many high-speed registers, and efficient I/O interfaces.

In the following sections, two currently available microprogrammed computers, the Interdata Model 4 of Interdata Corporation and the IBM 360/50 of IBM Corporation, will be studied in some detail to illustrate some differences. Among other things, they differ in their microprogramming philosophy.

Interdata 4

The Interdata 4 is a small-scale, multi-purpose computer with prewired, nondestructive read-only memory (ROM) of 400 nanosecond cycle time.[2] The logic of ROM is wired on a pluggable circuit card containing a 1024 word U-core ferrite transformer with wires winding through them to determine the contents of the store. It can be altered by rewiring with some special equipment.

There are four types of microprogram instruction format.[1,2] An instruction consists of 16 bits: 4 bits for the operation code field and 12 bits for the various fields according to the format. Figure 3 shows an example of the instruction format. The format resembles closely that of the conventional machine code instruction which provides easier microprogramming. A microprogram assembler as well as a simulator for testing microprograms is supplied and wiring is automatically done by a machine through the punched paper tape.

With this type of instruction format, it is easy to program but only one operation can be specified and overlapping of operations is not possible. This will cause a performance loss of the computer and is a considerable disadvantage over the IBM 360/50 which has a capability of specifying overlapping micro-operations.

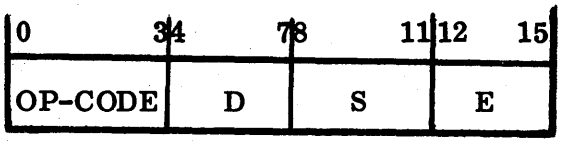


Figure 3—Interdata 4: Add, subtract, exclusive OR, AND, inclusive OR, and load instruction

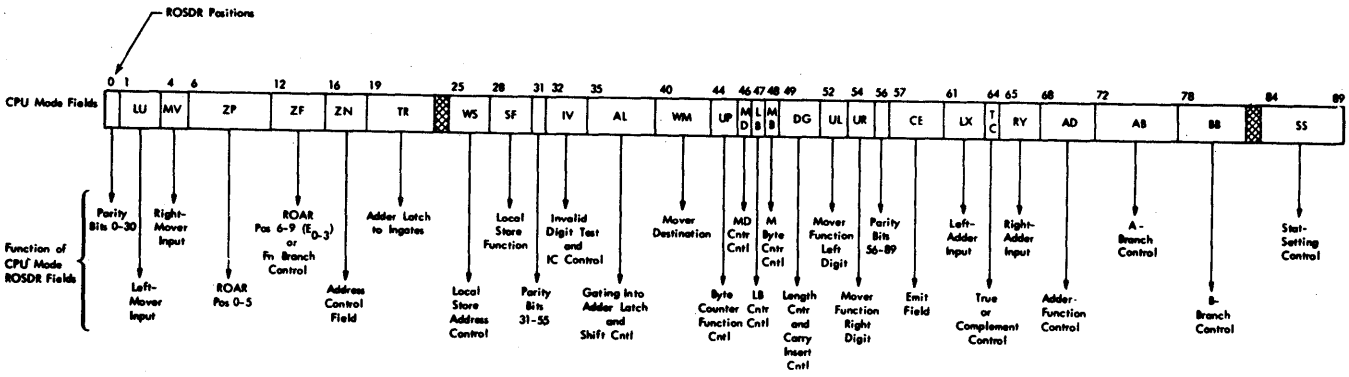


Figure 4—An IBM 360/50 micro-instruction

The significance of the Interdata 4 is that it is a user-microprogrammable computer with a limited facility for writing microprograms. It meets the requirements by providing a fast-read and very slow-write control memory.

IBM 360/50

The IBM 360/50 is a medium-scale, multipurpose computer. It operates with a 2815-word capacitor read-only store (CROS). Each word consists of 90 bits and controls the gates and control lines of the system for one 500-nanosecond machine cycle.

Unlike that of the Interdata 4 computer, the format of microprogram instructions is somewhat complex (Figure 4). A 90-bit instruction word is divided into a number of fields of various field length. Each field has a predetermined function. An instruction word is read out of ROS and set into the read-only storage data register (ROSDR) at every machine cycle. The address of the next micro-instruction is composed partly of the 90-bit micro-instruction in use and partly from the results of the current machine cycle. Specific bits from the ROSDR fields are combined with the CPU or I/O mode by the CROS-word decode logic to activate control lines. Therefore, a number of micro-operations are performed concurrently in one machine cycle, and the speed of machine performance is increased.

The Control Automation System (CAS) was developed by IBM as a programmer's assistance in microprogramming for the IBM 370. CAS accepts a listing of the micro-instructions prepared on the logic sketch sheet, produces the 90-bit pattern for each control word, and assigns the address for each ROS word. This reduces the microprogrammer's job to preparing the logic sketch sheet. It is not a simple task to prepare it, however, since there are fixed formats with which the microprogrammer must comply.

Before a microprogram is converted into the bit pattern for the ROS wiring, it is tested on a cycle-by-cycle simulation with various sets of initial conditions and traced step-by-step for the effects of instructions on the various components of the system. This will eliminate a system malfunction caused by ill-formed microprograms.

With these facilities, the pitfall of complex timing and gating restrictions is avoided and an effective use of the IBM 360 capabilities may be realized. Although it still seems somewhat cumbersome, the complexity of microprogramming is considerably reduced.

The two contemporary computers studied here are equipped with the writable ROS. Although their microprogramming and microprogram loading processes are still cumbersome, various equipments have been developed as an aide for easier microprogramming. Although, in principle, the concept of alterable microprogramming is observed in the philosophy of hardware design, manufacturers are still reluctant to relinquish microprogramming to general users.

COMMENTS ON ORGANIZATION

In spite of its flexibility, the microprogrammed computer is basically slow. The idea of two-level microprogramming discussed earlier suggests ways for improvement. The frequently-used management functions can be implemented by the lowest, gate-level microprograms. At this level, well-established speed-up techniques such as the microprogram memory interlacing, overlapping of various execution phases of a micro-instruction, and the exploitation of control parallelism will improve the efficiency of execution. These techniques, however, require detailed knowledge of processor timing and internal data-flows. The user will use the next higher level, i.e., minilevel, for his microprogram applications. At this level, instructions

are less machine dependent and, therefore, much easier to use.

If the computer has more than one microprogrammable, instruction execution units, the mode of their utilization will depend on the nature of computations. The microprogrammed control units would need access to common storage registers, for example, where parallel computations are performed on common, shared data areas. Synchronization between control units must also be established. Examples of such computations are automatic numerical error analysis, significance checking, performance measurements, and so forth.

Microprogrammed control had been successfully exploited in the newer designs of functional subsystems. The use of such special purpose units for instruction reformatting, address generation, data structure interpretation, and certain types of simple array processing may be suggested.

FINAL REMARKS

To conclude, further areas of actual and potential microprogram application are listed. These are not exhaustive, of course, but should lead the future microprogram users to further study of microprogram applications.

1. Compilers that map higher-level language source programs to microprograms via intermediate languages.
2. Control of time sharing systems (special macro-operations for data handling, queue manipulation, scheduling and allocation algorithms).
3. Interrupt servicing, queuing, status interpretation.
4. Facilitation of incremental compilers for time sharing.
5. Control of parallel computer organization.
6. Increased reliability through diagnosis of parts not being used in current instruction.
7. Increased ease and accuracy in fault detection through diagnostics on the microprogramming level.
8. Control of "degraded performance" capability for aerospace and medical applications.
9. Direct execution of process oriented languages.
10. Emulation of machine languages.
11. Image identification.
12. Cryptanalysis.
13. Control of associative memories (control of multiple operand, mask and results registers load and dump circuitry).
14. Information retrieval especially in connection with associative memories.

The possibility of microprogram application is limitless. As the microprogramming and hardware techniques advance, wider variety of possibility will be expected to develop. We hope that our study of user-microprogrammable computers will motivate the future microprogram user to further study the area of microprogramming and its applications. It is also our hope that this paper will incite those who are in charge of the design of fourth generation computer systems to consider the full impact of microprogramming to users.

PART TWO—TRADE OFF ALGORITHM

INTRODUCTION

The optimal allocation of resources to maximize computing throughput is one of the most important problems in computer design. The throughput of a computing system is a function of many parameters. One important problem in designing a microprogrammable computer system is the determination of the optimal size of the high-speed and expensive alterable microprogram memory as well as other types in a hierarchy of memories given the total resources allocated for memory. An algorithm for designing memory heirarchy should answer the following:

1. The optimum sizes of microprogram memory, and other components in a memory hierarchy in order to minimize the average access time for the user's activity profile for a given cost constraint. The optimum amount of information transfered to the microprogram memory.
2. The optimum set of memory types, including microprogram memory in a hierarchy with regard to a number of memory types, cost and access time.
3. The cost versus average access time tradeoffs for a memory hierarchy for a given activity profile. A change in the minimum average access time for an expenditure of some "x" dollars on the memory.

DEFINITIONS AND NOTATIONS

When a computer program is run, certain blocks of information are accessed more frequently than others. The activity profile of a given set of programs is the relative frequencies with which blocks of addresses are accessed when that set of programs is run.

The numbers of blocks accessed at frequencies F_1, F_2, F_3, . . . F_m are denoted by W_1, W_2, W_3, . . . W_m, respectively. Activity is defined directly proportional to the access frequency.

$$P_j = F_j \Big/ \sum_{i=1}^{m} F_i W_i \quad \text{where} \quad \sum_{i=1}^{m} P_i W_i = 1.$$

An m-dimensional vector $P = (P_1, P_2, \ldots P_m)$, where $P_i < P_{i+1}$, denotes the activities of a program. Associated with P, there is an m-dimensional vector W such that its ith component W_i represents the number of blocks of information accessed at activity P_i.

The activity profile then is defined by an ordered pair of vectors (P, W). In practice, it can be determined either analytically or by simulation and experimentation.[7,11]

Let us assume that n types of memory devices are available. Furthermore, let T_i denote access time and C_i denote the cost of *one* block of the memory type i.

With these definitions and notations, the problem is now stated.

STATEMENT OF PROBLEM

Given:

1) the maximum permissible cost G for the entire storage system;

2) n different types of memory available where the cost per block and average time to access one block are C_i and T_i respectively;

3) the activity profile of the information to be stored in the hierarchy is given by the $2 \times m$ matrix:

$$\begin{bmatrix} P_1, & P_2, & P_3, & \cdots & P_m \\ W_1, & W_2, & W_3, & \cdots & W_m \end{bmatrix}$$

where

$$P_i < P_{i+1} \quad \text{and} \quad \sum_{i=1}^{m} P_i W_i = 1;$$

determine the sizes of the microprogram memory as well as other memories and the location of information blocks in the storage such that

1) the total cost does not exceed G and

2) the average access time to any information block stored in the hierarchy is minimized.

Without losing generality, we may assume that G_0 is the cost of mass storage or Type O memory, and that it is one of the least cost and large enough to accommodate all the blocks in the program. We assume that one block of information occupies one unit of x memory space and a block is divisible between two memory types.

LINEAR PROGRAMMING FORMULATION

Let V_{ik} be the number of information blocks of activity P_i stored in memory type k, $(1 \leq k \leq n)$, and let

$$V_{i0} = W_i - \sum_{k=1}^{n} V_{ik} \tag{1}$$

so that the problem becomes

$$G_0 + \sum_{k=1}^{n} \sum_{i=1}^{m} C_k V_{ik} \leq G$$

$$\sum_{k=0}^{n} V_{ik} = W_i \quad \text{for} \quad i = 1, 2, \cdots, m$$

$$V_{ik} \geq 0 \quad \text{for} \quad 1 \leq i \leq m; \quad 0 \leq k \leq n \tag{2}$$

minimize

$$\bar{T} = \sum_{i=1}^{m} \sum_{k=0}^{n} P_i T_k V_{ik} \tag{3}$$

The size of memory type k denoted by U_k is then

$$\sum_{i=1}^{m} V_{ik} = U_k.$$

A set V_{ik} satisfying the condition for $1 \leq i \leq m$ and $1 \leq k \leq n$ is an optimal solution.

The following theorem allows the problem to be applied to a subset of all the available memory types.

THEOREM 1. Given three memory types, 1, 2, and 3, such that

$$C_3 > C_2 > C_1 \tag{4}$$

and

$$T_3 < T_2 < T_1 \tag{5}$$

and

$$\frac{T_1 - T_3}{C_3 - C_1} > \frac{T_1 - T_2}{C_2 - C_1} \tag{6}$$

for any activity profile, then there are no blocks of information stored in memory type 2 in an optimal solution.

The proof is straightforward. An interested reader may refer to Reference 12 for the proof.

ALGORITHM

By an application of Theorem 1, a new hierarchy is derived from the original memory hierarchy. While data require the same amount of microprogram memory space as that of non-microprogram memory space when transfered, note that a macro-instruction in non-microprogram memories is represented by several micro-operations when transfered into a microprogram memory. This means that the contents of non-microprogram memory require several times more space of the microprogram memory when transfered. This fact is reflected in the algorithm by the cost of memory.

Another consideration is that a macro-instruction is performed faster when it is represented by a microprogram and directly executed. Besides the high-speed of microprogram memory, this is largely due to an elimination of instruction fetch cycles at an execution time that an instruction is executed faster with a less number of machine cycles.

These two facts concerning a transfer of information from a non-microprogram memory to a microprogram memory must be reflected in the algorithm. Adjustments are made in deriving a new memory hierarchy, transfer values, and in the remainder of the algorithm.

(1) Determine memory type j such that for $0 < j \leq n$

$$\Delta_{0j} = \frac{T_0 - T_j}{C_j - C_0}$$

is maximum. Include the memory type j in the new derived hierarchy and eliminate memory types i for $0 < i < j$ from further consideration.

(2) Repeat the procedure described in (1) for the memory type k with $i \leq k \leq n$, replacing T_0 and C_0 by T_j and C_j respectively.

For the microprogram memory, i.e., the type n memory,

$$\Delta_{jn} = \frac{T_j - RT_n}{SC_n - C_j}$$

where R and S are multiplying factors to adjust the access time T_n and cost C_n of the microprogram memory, respectively.

(3) From the new hierarchy, determine transfer values X_j.

$$X_j = \frac{T_{j-1} - T_j}{C_j - C_{j-1}} \qquad j = 1, 2, \cdots n'-1$$

where n' is a number of memory types in the new hierarchy.

To determine the transfer value for the microprogram memory,

$$X_{n'} = \frac{T_{n'-1} - RT_{n'}}{SC_{n'} - C_{n'-1}}$$

where $T_{n'}$ and $C_{n'}$ are the access time and cost of the microprogram memory (the type n' memory in the new hierarchy).

(4) Determine transfer priorities Z_{ij} where

$$Z_{i,j} = P_i X_j \qquad i = 1, 2, \cdots m, \quad j = 1, 2, \cdots n'$$

(5) Order the $Z_{i,j}$ in decreasing order of magnitude. Let

$$Y_1 = \max_{ij} Z_{i,j} = Z_{i(1),j(1)}$$

$$Y_2 = \max_{ij} Z_{i,j} = Z_{i(2),j(2)} \text{ where } i, j \neq i(1), j(1)$$

$$Y_3 = \max_{ij} Z_{i,j} = Z_{i(3),j(3)} \text{ where } i, j \neq i(1), j(1) \text{ and } i, j \neq i(2), j(2)$$

. . .
. . .
. . .

(6) The transfer implied by $Z_{i,j}$ is the shift of all information of activity P_i from memory type $j-1$ to memory type j; that is, (i) decrease the size of memory type $j-1$ in the existing hierarchy by W_i unit of memory space, (ii) increase the size of memory type j in the existing hierarchy by W_i units of memory space, and (iii) shift W_i blocks of activity P_i from memory type $j-1$ to the vacant space created in memory type j. Note that when information in nonmicroprogram memory is transfered to a microprogram memory it is expanded into a representing microprogram.

Let $c(k)$ and $t(k)$ be the cost and average access time of the hierarchy after the kth transfer. Let $\Delta c(k)$ and $\Delta t(k)$ be the amount of change in the cost and average access time due to the kth transfer. Then

$$c(k) = c(k-1) + \Delta c(k)$$
$$t(k) = t(k-1) + \Delta t(k) \qquad k = 1, 2, 3\cdots$$

where

$$\Delta c(k) = \begin{cases} W_i(C_j - C_{j-1}) & \text{for some } i \text{ and } j \neq n' \\ W_i(SC_j - C_{j-1}) & \text{for some } i \text{ and } j = n' \end{cases}$$

$$\Delta t(k) = \begin{cases} -W_i P_i(T_{j-1} - T_j) & \text{for some } i \text{ and } j \neq n' \\ -W_i P_i(T_{j-1} - RT_j) & \text{for some } i \text{ and } j = n' \end{cases}$$

$c(0) = G_0$ and

$t(0) = T_0$.

it is assumed that all information is stored in the mass memory in the initial state, i.e., the initial cost is G_0 and access time T_0. Determine $c(k)$ and $t(k)$, and plot $c(k)$ versus $t(k)$ for $k = 0, 1, 2 \ldots$. Note that adjustments are made to Δc and Δt when information is transfered to a microprogram memory. The plot is the cost-time characteristic. (Figure 6)

Note that steps (4), (5), and (6) must be repeated to obtain the cost-time characteristic for each given activity profile.

The optimal microprogram memory size in the optimal memory hierarchy and the location of information blocks in the hierarchy

Determine k' such that

$$c(k') \leq G \leq c(k'+1).$$

If $c(k) \leq G$ for all k, then store all the information in the fastest microprogram memory. The optimal size of microprogram memory is

$$U_n = \sum_{i=1}^{m} W_i$$

and the size of memory type k, $k \neq 0$ and $k \neq j$, is zero, i.e., the microprogram memory alone constitutes the optimal memory hierarchy.

If there exists a k' such that inequality (9) holds, let the ordered transfer priorities $Y_1, Y_2, \ldots, Y_{(k+1)}$ be $Z_{i(1)j(1)}, Z_{i(2),j(2)} \cdots Z_{i(k'+1),j(k'+1)}$. To determine the location of information of activity P_u inspect the ordered transfer priorities from right to left until $Z_{i(q)j(q)}$ such that $i(q) = u$ and $i(r) \neq u$ for $k'+1 \geq r \geq q$ is found.

(i) If no such $Z_{i(q)j(q)}$ exists, all the blocks of information with activity P_u are stored in the mass memory.

$$V_{u0} = W_u$$
$$V_u^k = 0 \qquad \text{for} \quad k \neq 0$$

(ii) If such a $Z_{i(q)j(q)}$ exists then the number of blocks of information with activity P_u stored in memory type j' where $j' = j(q)$ is

$$V_{uj'} = \begin{cases} \dfrac{G - c(k')}{C_{j'} - C_{j'-1}} & \text{if} \quad q = k'+1, \\ W_u & \text{if} \quad q \neq k'+1 \end{cases}$$

The remaining blocks of activity P_u are stored in memory type $j' - 1$

$$V_{u(j'-1)} = W_u - V_{uj'}$$
$$V_{uk} = 0 \qquad \text{for} \quad k \neq j' \qquad \text{and} \quad k \neq j'-1$$

The size of memory type k in an optimal memory hierarchy is

$$U_k = \sum_{i=1}^{m} V_{ik} \qquad \text{for} \quad k = 1, 2, 3 \cdots n'.$$

It can be shown that the algorithm described above is feasible and optimal. However, it is stated as a theorem and interested reader should refer to Reference 12 for the proof.

THEOREM 2: The Algorithm is feasible and optimal.

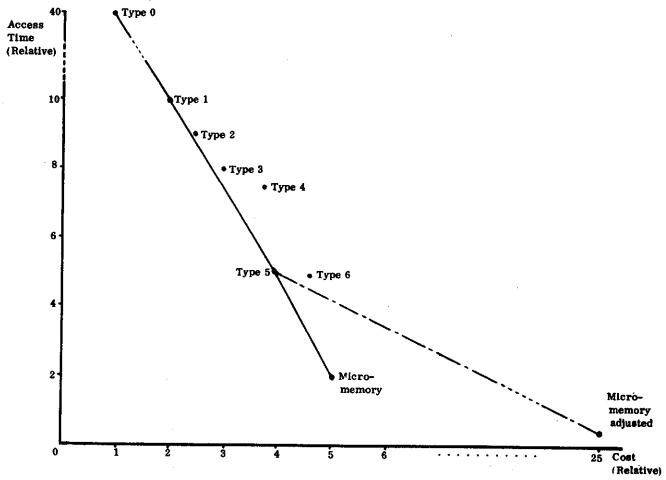


Figure 5—Cost/time plot for determining the new hierarchy

EXAMPLE

Given maximum possible cost $G = 425$, a mass memory of cost $G_0 = 90$ and average access time $T_0 = 40$. The number of available memory types $n = 7$ with the following specifications, where the type 7 memory is a high-speed control memory.

Memory type	0	1	2	3	4	5	6	7
Access time T_i	40	10	9	8	7.5	5	4.9	1.2
Cost/unit memory space C_i	1	2	2.5	3	3.8	4	4.6	5

The activity profile of the given program is as follows:

Activity P_i	0.0082	0.0128	0.0209	0.0046	0.0018	0.012	0.0143	0.0256
No. of blocks w/this activity W_i where $P_i W_i = 1$.	10	5	20	15	20	5	10	5

Throughout this example we assume that a macro-instruction in a noncontrol memory is, on the average, represented by six micro-operations when transfered to a control memory and executed in ⅓ of the instruction execution time. Data transfered from a non-microprogram memory take the same amount of microprogram memory. Therefore, we assume that information in one block of non-microprogram memory will be expanded on the average into five microprogram memory blocks. This assumption is expressed in the algorithm by:

$$R = \tfrac{1}{3}$$

and

$$S = 5.$$

Solution

1. Obtain the new memory hierarchy from the original hierarchy. (Figure 5) Values in parentheses are the adjusted access time and cost of control memory.

Original Hierarchy No.	0	1	5	7
New Hierarchy No.	0	1	2	3
Access time	40	10	5	1.2(0.4)
Cost/unit memory space	1	2	4	5(25)

2. Transfer priorities

$$X_1 = 30, \quad X_2 = 2.5, \quad X_3 = 0.22$$

3. Priority matrix:

$$Z = \begin{bmatrix} 0.246 & 0.0205 & 0.001804 \\ 0.384 & 0.032 & 0.002816 \\ 0.627 & 0.05225 & 0.004598 \\ 0.138 & 0.0115 & 0.001012 \\ 0.054 & 0.0045 & 0.000396 \\ 0.36 & 0.03 & 0.00264 \\ 0.429 & 0.03575 & 0.003146 \\ 0.768 & 0.064 & 0.005632 \end{bmatrix}$$

TABLE I—Average Access Time and Total Cost Associated with Transfers of information.

	Type 0 Amount transfrd		Type 1 Amount transfrd		Type 2 Amount transfrd		Type 3 Amount transfrd		Total Cost ΔC_i	Total Cost C_i	Avg. Access time ΔT_i	Avg. Access time T_i
		90		0		0		0		90		40
Y_1	- 5	85	5	5		0		0	5	95	- 3.84	36.16
Y_2	-20	65	20	25		0		0	20	115	-12.54	23.62
Y_3	-10	55	10	35		0		0	10	125	- 4.24	19.33
Y_4	- 5	50	5	40		0		0	5	130	- 1.92	17.41
Y_5	- 5	45	5	45		0		0	5	135	- 1.8	15.61
Y_6	-10	35	10	55		0		0	10	145	- 2.46	13.15
Y_7	-15	20	15	70		0		0	15	160	- 2.06	11.09
Y_8		20	- 5	65	5	5		0	10	170	- .64	10.45
Y_9	-20	0	20	85		5		0	20	190	- 1.08	9.37
Y_{10}		0	-20	65	20	25		0	40	230	- 2.09	7.28
Y_{11}		0	-10	55	10	35		0	20	250	- .715	6.565
Y_{12}		0	- 5	50	5	40		0	10	260	- .32	6.245
Y_{13}		0	- 5	45	5	45		0	10	270	- .3	5.945
Y_{14}		0	-10	35	10	55		0	20	290	- .41	5.535
Y_{15}		0	-15	20	15	70		0	30	320	- .345	5.190
Y_{16}		0		20	- 5	65	5x5	25	105	425	- .5888	4.6012
Y_{17}		0		20	-20	45	5x20	125	420	845	- 1.9228	2.6784
Y_{18}		0	-20	0	20.	65		125	40	885	- .18	2.4984
Y_{19}		0		0	-10	55	5x10	175	210	1095	- .6678	1.8306
Y_{20}		0		0	- 5	50	5x5	200	105	1200	- .2944	1.5362
Y_{21}		0		0	- 5	45	5x5	225	105	1305	- .276	1.2602
Y_{22}		0		0	-10	35	5x10	275	210	1515	- .3772	.8830
Y_{23}		0		0	-15	20	5x15	350	315	1830	- .3174	.5656
Y_{24}		0		0	-20	0	5x20	450	420	2250	- .1656	.4000

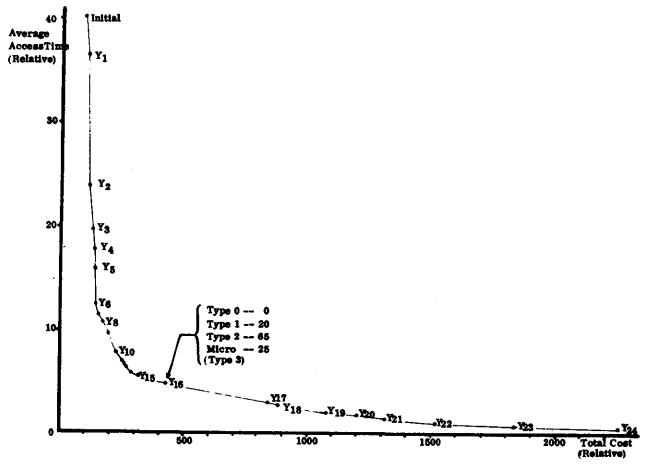


Figure 6—Cost/time characteristics for determining optimum Microprogram memory and average time

4. From the elements of priority matrix Z, obtain ordered transfer priorities:

$Y_1 = Z_{81}$	$Y_9 = Z_{51}$	$Y_{17} = Z_{33}$
$Y_2 = Z_{31}$	$Y_{10} = Z_{32}$	$Y_{18} = Z_{52}$
$Y_3 = Z_{71}$	$Y_{11} = Z_{72}$	$Y_{19} = Z_{73}$
$Y_4 = Z_{21}$	$Y_{12} = Z_{22}$	$Y_{20} = Z_{23}$
$Y_5 = Z_{61}$	$Y_{13} = Z_{62}$	$Y_{21} = Z_{63}$
$Y_6 = Z_{11}$	$Y_{14} = Z_{12}$	$Y_{22} = Z_{13}$
$Y_7 = Z_{41}$	$Y_{15} = Z_{42}$	$Y_{23} = Z_{43}$
$Y_8 = Z_{82}$	$Y_{16} = Z_{83}$	$Y_{24} = Z_{53}$

5. From the ordered priorities we obtain

ΔC_i and ΔT_i for $i = 0, 1, 2, \ldots$.
C_i and T_i for $i = 0, 1, 2, \ldots$.

Table 1 shows the amount of information transfered between memories and its effects in the total cost and average access time. From the table, T_i and C_i are plotted (Figure 6) to show the minimum average access time versus total cost curve when the program is stored in and executed from the optimized memory hierarchy. The sizes of each memory type at certain cost are computed using the LP program developed earlier and indicated in Table 1 and in Figure 6.

Results

The result is shown in Table 1 and in Figure 6. Given the maximum permissible cost $G = 425$, seven types of memories with the initial cost of mass memory = 90, and average access time = 40, the optimum size of the microprogram memory is 25 blocks. The result also shows that the optimum sizes

of the memory types 2, 1, and 0 in the new hierarchy are 65 blocks, 20 blocks, and 0 blocks, respectively. The optimum memory combination costs 425 units and its average access time is 4.6012.

ACKNOWLEDGMENT

The Authors are grateful to Mr. Joel Shechter for many suggestions that led to the improvement of the paper.

Specification of Microprogrammed Computer Available

Feature	*IBM 360/50*	*Micro 800*	*Interdata 4*	*IC-9000*
Control memory type	Capacitor read-only	Discrete Diode Read-only	Ferrite transformer read-only	Read/write
Micro memory access time	500 nsec.	220 nsec.	400 nsec.	500 nsec.
Micro Instr. word length	90 bits	16 bits	16 bits	32 bits
Micro memory capacity	2816 wds.	256 wds.	512 wds.	
Micro Instr. format	bits/function	machine code type	machine code type	machine code type
Overlapping micro-ops	Yes	No	No	Yes
Single Cycle mode	No	Yes	No	Yes
User Alterable (indirect)	No	No	No	Yes
User Alterable (direct)	No	No	No	Yes
Simulator software	Yes	Yes	Yes	Yes
Factory Wired micro memory	Yes	Yes	Yes	Yes
Indexing micro memory	No	No	Yes	Yes
Indexing in microprocessor	No	No	Yes	Yes
Binary add (*w/o* shift)	500 nsec. (32 bits)	220 nsec. (8 bits)	800 nsec. (16 bits)	500 nsec. (36 bits)
Associative registers	No	No	No	No
Interrupt thru micro prog.	Yes	Yes	Yes	Yes
Micro level diagnostic	No	Yes	Yes	Yes

REFERENCES

1 *Model 4 micro-instruction reference manual*
Interdata Pub No 29–032

2 *Micro-code programming of a read-only memory computer*
Interdata application No 103 and No 38–020

3 *Introduction to principles of operation—Standard model 9*
Inner Computer Form 1006–1

4 A D FALKOFF K E IVERSON E H SUSSENGUTH
A formal description of System/360
IBM Systems Journal Vol 3 No 3 pp 198 1964

5 S S HUSSON
Microprogramming manual for the IBM system/360 model 50
Technical report TR00 1479–1 October 1967

6 H W LAWSON
Programming-language-oriented instruction stream
IEEE Transactions on Computers, pp 476–485 May 1968

7 J MARTIN
Programming real time computer systems
Prentice Hall Englewood Cliffs New Jersey 1965

8 A OPLER
Fourth generation software
Datamation Vol 13 No 1 pp 22 January 1967

9 D F PARKHILL
The challenge of the computer utility
Addison–Wesley Reading Massachusetts 1966

10 L L RAKOCZI
The Computer-within-a computer, a fourth generation concept
IEEE Computer Group News Vol 3 No 2 pp 14 1969

11 C V RAMAMOORTHY
Markov analysis of computer programs
Proceedings of National Meeting of ACM 1965

12 K M CHANDY C V RAMAMOORTHY
Optimization of information storage systems
Information and control Vol 13 pp 509–526 December 1968

13 R F ROSIN
Contemporary concept of microprogramming and emulation
Computing Surveys Vol 1 No 4 pp 197–212 1969

14 W Y STEVENS
The structure of system 360' Part II—System implementation
IBM Journal Vol 3 p 136 1964

15 M V WILKES
The best way to design an automatic calculating machine
Manchester University Computer Inaugural Conference p 16 1951

16 M V WILKES
Microprogramming
Proc Easter Joint Computer Conference p 18 December 1958
17 M V WILKES et al
The design of a control unit of an electronic digital computer
Proc IEEE Vol 105 p 121 1958
18 M V WILKES
The growth of interest in microprogramming A literature survey
Computing Surveys Vol 1 No 3 pp 139–145 September 1969

Firmware sort processor with LSI components

by HARUT BARSAMIAN

National Cash Register Company
Hawthorne, California

INTRODUCTION

With the latest achievements of the large scale integration (LSI) technology, a major qualitative breakthrough in the information processing art is expected. The consensus in the computer industry is that the next generation of computers will be constructed with LSI components.

Three inherent characteristics of LSI, subject to incessant improvement, appeal most to both computer manufacturers and users. First, significant reduction in the production cost due to the automated design and fabrication of large and complex logic circuits in a single package. Second, improvement of the overall system reliability by at least one order of magnitude because of the automated manufacturing processes and sharp reduction of the number of external interconnections. And, last, higher operational speeds because of microminiaturization and advanced technology. Admitting however these potential capabilities of LSI and its inevitable impact on the future of the computer industry, it should be emphasized that the introduction of the new generation of computers with LSI components cannot repeat the past pattern of computer generations because of several factors:

a. The qualitative superiority of transistors and later IC's compared with their predecessors were indisputable from the beginning. The mass production and the absorption of these components by the computer manufacturers went smoothly and fast. Yet LSI has not achieved a similar status. The problems of power dissipation and cooling, packaging and interconnections, yield and mass production of off-the-shelf products are of a much higher magnitude for LSI than were those of its predecessors.
b. For the full utilization of LSI's expected capabilities, new concepts and techniques should be developed for all phases of computer design. A conventional computer logic simply redesigned with LSI components can hardly be described as a new generation of computers, nor can it disclose any significant technical and economical advantages. It is estimated that even if LSI technology were offered free, such redesign would reduce the hardware cost by only 3.5 percent.[1]
c. The multibillion dollar investments made by computer manufacturers and users in the current information processing installations will retard an abrupt growth of a new generation of computers.

These factors indicate that the introduction of a new generation of information processing systems will be a slow and evolutionary process. There will be a transitionary computer generation, that combines basic third generation equipment and operating software and new special purpose processors, peripheral controllers and terminals designated to increase the overall computations per dollar of the present computers. These developments will overcome the inefficiencies and/or limitations of the current computer generation, thereby substantiating the economical as well as technological grounds for a truly new generation of computer systems.

The state-of-the-art of the semiconductor technology confirms that LSI can make a decisive contribution in this evolutionary trend.

Background

A brief survey of the computer industry's development history reveals that the continuously improving cost-performance index of semiconductor and magnetic circuit components stimulated the creation of larger and faster CPU's (including the main memories). This

factor, as well as the versatile growth of the computer applications, necessitated the development of the sophisticated software systems and programming techniques. Meanwhile the peripheral devices, electromechanical and optical, remained quite primitive in their performance, requiring substantial software support from the system.

As a consequence of these developments in a typical computer installation, costs for software and applications programming reach as high as 70 percent of the total expenditure. With the advance of component technology this trend may soon result in a hardware/software cost split of 15 to 85 percent (the CPU sharing only 3 percent of the hardware cost).

Objectives

It seems apparent that basic changes in the computer architecture and redistribution of the hardware and software resources within the system, with much more emphasis on hardware functions, must be accomplished. Accordingly, the following basic design objectives must be formulated:

a. Minimize the software sector of the system by performing certain control procedures and standard routines by hardware and/or firmware.
b. Incorporate sufficient local logic and self-control into the peripherals and terminals so they can be driven by the computer without device-specific software routines.
c. Modularize hardware and software, permitting the tailoring of efficient systems for a particular application.

The concept

One of the principal means for achieving these objectives is the decentralization of the computer's processor tasks. Special purpose hardware/firmware processors are substituted for program routines and/or entire algorithmic functions that have ordered structures and frequent use in a large spectrum of applications.

The ultimate purpose of such trade-offs is to achieve the following results:

a. Release more CPU space and time for other, more specialized jobs and supervisory functions.
b. Simplify the software control of the whole computing system.
c. Ease applications programming.
d. Increase the system's overall throughput.

These special purpose processors must contain adequate self-control to perform their functional duties with minimal control support, and must not cause any significant hardware or software modifications in the system.

For multiprogramming and time sharing systems, this approach can improve the overhead by decreasing the heavy information traffic within the system and the frequent switching of the CPU from one task to another.

This paper suggests an implementation of the proposed concept. It discusses a special purpose processor designed with LSI components, dedicated for sorting.

THE SORT PROCESSOR

The sort routine

The purpose of computer sorting is to generate a systematically ordered data file from randomly accumulated raw data made up of fixed or variable length records. The sort routine organizes these records in ascending or descending order of their key words which identify each record.

Because of the limited size of the computer's main memory, the sorting is performed in two phases:

a. Internal sort, generating strings of sorted records.
b. Merge, when the strings are merged into single systematically ordered file.

To accomplish sorting, multiple passes of the internal sort and merge phases are executed. These are interleaved by I/O routines that transfer data between the main memory and the peripheral file (tapes or discs). In both of these phases the purpose of the sort routine is the same, to compare the keys and to order the records in a desired sequence.

Three characteristics of the sorting operation (traditionally a software function) make it ideally suited to a firmware implementation:

Frequent use

Sorting is among the basic algorithmic functions of almost any computer installation. It has applications in commercial and scientific problems, operating systems, program assembling and compiling, and in list processing. It is estimated that sorting comprises from 25 to 50 percent of the overall computer workload for business oriented systems.

Ordered structure

The sort routine is supervised by the sort-merge control program. It is called after the raw data is loaded into the Main Memory (MM) and boundary conditions of the work and buffer areas are defined. The sort routine iterates, comparing keys and readdressing or relocating the records in the required order until the end of the current string is reached. When this condition occurs the sort routine transfers control back to the sort-merge control program. The sort routine is called again after the current sorted string is transferred to the peripheral file, new raw data is loaded into the MM and new boundary conditions are set. This process continued until the sort operation is completed. Thus, the sort routine does not have a complex structure in the sense of multiple branches and linking points with the other system components. It interfaces with the sort-merge control program with one entry and one exit, and is monotonously iterative by its nature.

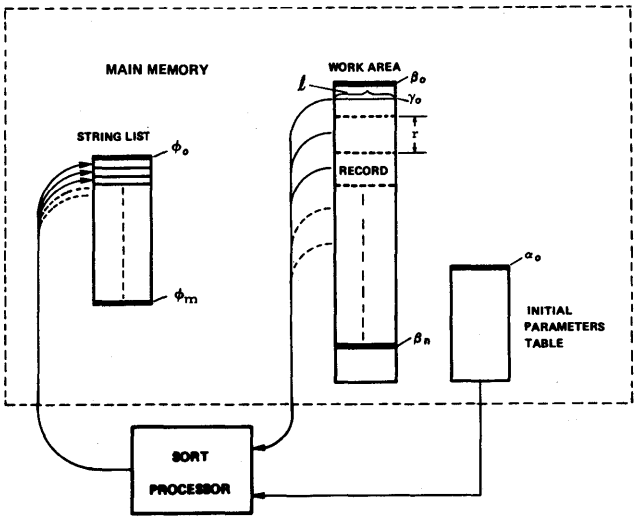


Figure 2—Memory allocation and data flow

Time consuming

The CPU time for the sort routine to generate a sorted string from a random set of records is consumed by two basic functions: comparison of the keys and, accessing the MM for reading the keys and compiling the list of the string. The number of these functions, and consequently the consumed CPU time, will increase drastically if the length of the keys are greater than the length of the machine word and this is the case for the majority of sorting jobs.

The analysis of the benchmark problems on sorting reveals that up to 40 percent of the total sorting time (see later section of this paper) is for CPU operations.

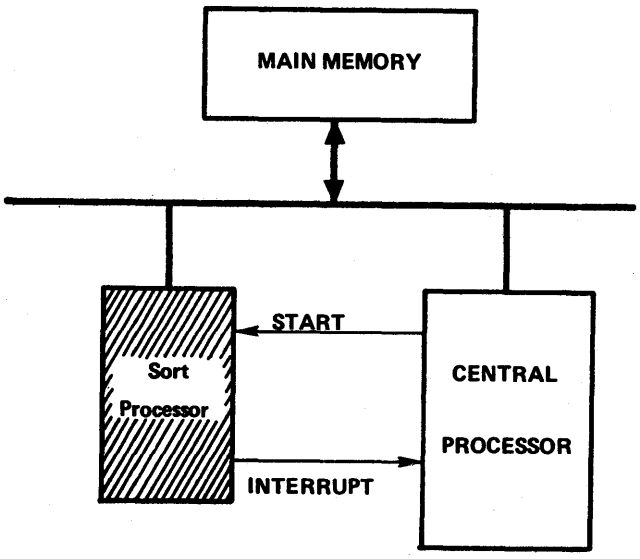


Figure 1—System configuration

The functional description

The Sort Processor (SP) is an internally programmed, firmware special purpose processor dedicated for performing the sort routine outside of the computer's CPU. The SP shares the common MM with the CPU on a lower priority basis and has the simplest interface with the CPU (Figure 1).

The START signal informs the SP that a Control Word (CW) is available on the MM bus. The CW consists of function and address fields. The function code indicates the type of operation to be performed by the SP, e.g., sort (ascending or descending), transfer status, resume, terminate. The address field specifies the starting address (α_0) of the initial parameters and boundary conditions table required for the sorting. This table set by the sort-merge control program, contains the following parameters (Figure 2):

> β_0 and β_n are the initial addresses of the first and the last records in the work area.
> ϕ_0 and ϕ_m are the initial and final addresses of the string list buffer.
> (Instead of β_n and ϕ_m the number of records in the work area (n) and the size of the string list (m) can be given.)

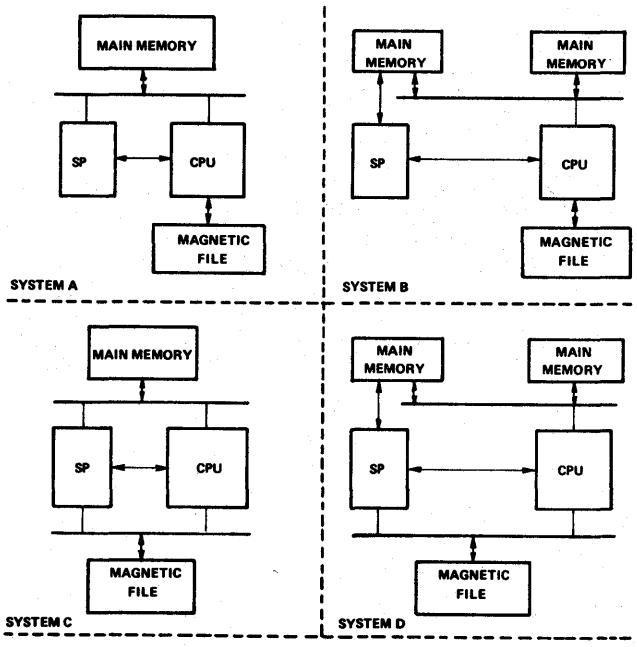


Figure 3—System configurations with sort processor

γ_0 = address of the key word of the first record
l = length of the key
r = length of the record (considering fixed length of records)

The operating sequence of the SP, after receiving the CW, typically proceeds in the following manner:

1. Reads the initial parameters table by the address α_0 and stores it in its Register File (RF).
2. Generates the effective addresses of the consecutive keys by computing $\gamma_i = \gamma_0 + ir$ starting with $i = 0$, subsequently $i = 1, 2, 3 \ldots, n$.
3. Reads the key (K_i). Meanwhile generates the initial addresses of each record $(\beta_i = \beta_0 + ir)$, and stores the codes $K_i\beta_i$ in the consecutive locations of its Search Memory (SM). The capacity of the SM can be smaller than the number of records in the work area, so that in the initial load the SM will be filled at $i < n$.
4. Locates the first desired (the highest or lowest) key (K^0) from the SM and stores the corresponding address (β^0) in the ϕ_0 location.
5. Replaces the vacancy in the SM with the next $(K_{i+1})(\beta_{i+1})$ code available from the work area.
6. Searches for the next desired key using the key K^{j-1} located at the previous $(j-1)$ cycle as base key for the comparison, and stores the address β^j corresponding to the newly located K^j into the location ϕ_j, and returns to step 5.

The iteration of steps 5 and 6 continue until the end of the string. This may occur when one of the following conditions arise:

a. $\beta_i = \beta_n$ (or $i = n$) indicating that the work area is exhausted.
b. $\phi_j = \phi_m$ (or $j = m$) indicating that the string list is exhausted.
c. No more successful searches in the SM are possible. The K^j was the last desired key, e.g., the SM does not contain any more keys that are greater than (or less than) the current base key.

Now the SP interrupts the CPU and stays idle until a new control word is received.

The SP continues functioning if it is preassigned to control the peripheral file (Figure 3, Systems C and D), and may:

a. transfer the sorted string (by the string list) from the MM into the peripheral file, load new raw data into the MM;
b. reorganize the memory map, move the buffer areas;
c. exchange status information with the Systems Supervisor, and resume sorting operations.

The degree of the complexity of these control functions depends upon the computer systems architecture and the preestablished functional duties of the SP.

Although the algorithm described appears to be optimum for the proposed systems organization, other sorting methods can be implemented.

The hardware structure

The SP consists of three major functional blocks, all designed with MOS LSI components: search memory, register file and microprogram storage. The block diagram of the SP is illustrated in Figure 4.

Search Memory (SM)

The SM is divided into two sectors. The KEY sector, designated for storing and searching key words, includes logic for comparing keys. The ADDRESS sector stores the initial addresses of the records. The number of bits per word in this sector equals $\log_2 M$, M being the size of the computer's MM in words directly accessible to the SP. Both these sectors are independently expandable in their bit directions, and the whole SM is expandable in the word direction in modules. These capabilities allow the SP to meet various sorting applications and to be integrated in computer systems that have different MM sizes.

Two types of MOS LSI memories for the SM are considered:

a. Associative Memory (AM). A modular AM with LSI components can be organized, using monolithic or hybrid technology. The logic to perform the "next greater" and "next smaller" functions is integrated into the AM chips. This could allow the SM to locate the next desirable key in an interrogation cycle. For current MOS technology, this is in the range of a few microseconds. However, the integration of these functions, because of their complexity, would result in a low yield of the AM chips. Also, because of the specific nature of these functions, such an AM chip might have limited marketplace. For these reasons, the integration of only the "equality" function appeared to be a more reasonable approach. To locate the next desired key in an "equality" search, the current key is modified (incremented or decremented) and compared continuously until an "equality" response is detected. The average number of these comparisons is equal to one-half of the key length in binary bits. Such a sacrifice in the searching speed seems to be justifiable by the economic reasons mentioned.
b. Recirculating Memory (RM). The RM is organized with recirculating MOS dynamic shift registers. The initial key is compared with the contents of the RM for "equality." After the response is detected, the initial key is modified and compared continuously until the next desired key is detected. The search is performed by comparing sequentially each word in the RM with the base key using a single comparator for the whole RM, as opposed to the AM, which contains a comparator per each word. The search time for the RM depends not only on the length of the key word, but also on the length of the shift registers composing the RM.

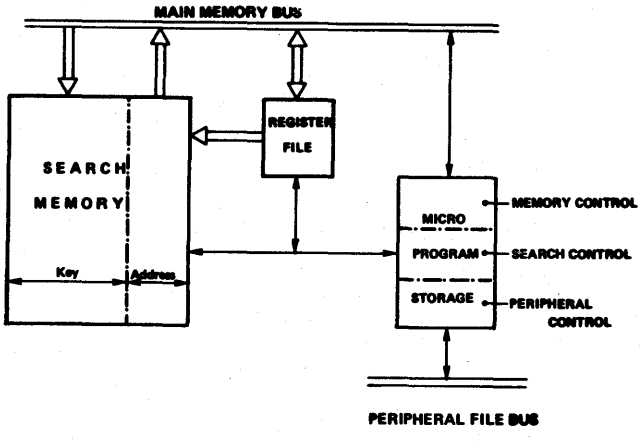


Figure 4—Sort processor block diagram

TABLE I—Comparison table for search memories

Type of SM	MOS Components Per Cell	I/O Pins Per Cell Ratio	Relative Search Time
AM	10	1:4	1
RM–1	6	1:13	5
RM–2	6	1:18	20

Based on the state-of-the-art of the MOS technology, a comparison is made between the AM utilizing the "equality" function only and two types of RM: the RM–1 with dual 64 bit, and the RM–2 with 256 bit dynamic shift registers. The frequency range (4 MC) and the other electrical characteristics of the chips are identical for both RM–1 and RM–2. The comparison is made for a 256 word SM, with the length of the keys in eight bytes (characters). The RM–1 has four external comparators functioning in parallel, one for each 64 word module, while the RM–2 has one external comparator only for the whole 256 word module.

The results of this comparison are summarized in Table I. It is evident from this comparison that RM's offer slower performance at lower cost. The worst case search time of the RM's can be estimated by the following formula:

$$T_{SR} = \left(\frac{W \cdot b}{2f} \times 10^{-6}\right) \text{sec}$$

where: W = number of words in the RM module (or the length of the shift register) consisting of a single comparator
b = length of the key in binary bits
f = frequency of the shift register in megacycles

The search time can be decreased by using shorter shift registers, if the cost for the additional hardware is justified. More dramatic improvements are achievable through the increase of the frequency (f). The current MOS technology already reaches up to the 20 MC range for the shift registers. Further improvements are expected in the characteristics of MOS associative memories and shift registers.

SM's of various searching speeds can be organized for a given cost-performance criteria. However, for the SP as a low priority background processor in a

computer system, the critical issue seems to be the economical factor, not the inherent speed. Accordingly, the use of dynamic shift registers for the SM presently appears to be more reasonable.

Register File (RF)

This is a scratch pad memory used to store initial parameters and boundary conditions as described earlier. Several temporary storage registers for indexing and counting are also included in the RF.

To make the RF more uniform and functionally flexible, all registers have the same $\log_2 M$ length. Sixteen registers in the RF appear to be sufficient.

Microprogram Storage (μS)

This random access memory stores the microprogram of the SP. Either read-only (ROM) or read-write (RWM) memories can be used for the μS. Comparing the ROM vs RWM, the following factors must be considered:

a. Because of the nondestructive read-out of the semiconductor memories, the ROM does not offer significant speed or economical advantages.
b. A higher yield can be achieved in a ROM of a given size substrate. However, this may not result in a decisive advantage because the production of ROM requires a certain degree of customization, while the RWM is an established off-the-shelf product.
c. The RWM allows greater flexibility and simplicity in microprogram alterations, debugging and maintenance.

Thus, the LSI RWM for the μS appears to be more desirable. Now the SP can perform not only various sort algorithms but also complementary functions such as table look-up, file maintenance, and list processing by simple reloading the μS by the desired microprogram.

Three basic microroutines reside in the μS:

a. the search microroutine which controls the SM and generates the MM addresses of the sorted records,
b. the MM interface control routine which performs all the communications between the MM and the SP,
c. the peripheral file interface control routine for controlling the I/O operations if the SP needs to communicate directly with the peripheral file (see Figure 3, System Configurations C and D).

Each of the microroutines are stored in individual LSI memory units and monitored by a common synchronizer. Such a partitioning of the control medium allows:

- Simultaneous execution of the search and interface control microroutines.
- More efficient use of LSI technology.
- Easy integration of the SP with almost any computer system by simply altering the microroutines.
- Easier maintenance and diagnostics.

From the economical standpoint, this approach does not cause any cost increase. Unlike magnetic memories, the size of the LSI semiconductor memory does not affect the bit price. Roughly 4096 bits of RWM, organized 128 × 32, are required for the μS.

SYSTEM CHARACTERISTICS AND CONFIGURATIONS

The SP as a "black box" can be integrated practically with any computer system and relieves the computer's CPU of the burden of sorting. It is applicable for computing systems operating in different processing modes. In the conventional batch processing systems, the SP functions as a stand-alone, low priority processor. In real-time or time-sharing systems, the SP functions as a background in-house processor. Substituting for the sort routine only, the SP does not cause any structural changes in the computer system architecture.

The system characteristics of the SP are summarized as follows:

a. The SP is easily connected to the MM channel of the computer and does not require any specific and/or additional hardware provisions from the computer (it behaves as any peripheral controller).
b. In a multiprocessing environment, the SP shares the common MM with the other processing units on a preestablished priority basis.
c. The interface between the SP and the MM is asynchronous and operates on the request-acknowledgement basis.
d. The SP requires the simplest software support. Statements like SORT ASCENDING, SORT DESCENDING, RESUME, TERMINATE, TRANSFER STATUS on the systems language level must be compiled into a single control word format which sets the SP to the appropriate operational state. Further, the SP performs the specified function autonomously.
e. The sort-merge control program performs overall supervision and interaction of the SP with the

system. Data preparation and MM allocation required for sorting also can be performed.

The SP can be integrated with the computer system in several configurations. Four typical system configurations are illustrated in Figure 3, and are described as follows:

SYSTEM A is the simplest configuration where the SP shares the MM with the CPU on a lower priority basis. The sorting time for this configuration is relatively long.

SYSTEM B allows the SP more freedom in accessing the appropriate MM bank. Although the SP remains a low priority processor, this configuration results in higher sorting speed.

In both system configurations A and B, data transfer between the MM and the peripheral file for sorting is accomplished through the conventional I/O channel and is controlled by the appropriate software routine.

SYSTEM C has the same MM sharing scheme as System A, in addition, the SP shares the peripheral file with the CPU. The control of the data exchange between the MM and the peripheral file, required for the sort-merge operations, is performed by the appropriate microroutine of the SP.

SYSTEM D combines the MM sharing scheme of System B and the peripheral file sharing scheme of System C. System D comprises fully parallel processing capabilities and offers the highest sorting efficiency.

In all of these configurations, the logic structure and the basic functional blocks of the SP remain practically unchanged. The specific interface characteristics of Systems B, C and D are easily programmed into the microprogram storage. The choice of a configuration depends upon the applications spectrum of the given computer system. Configurations C and D seem to be more applicable for the business computer systems where large amounts of data are to be processed and the I/O portion of the sort-merge operations are of significant magnitude. Configurations A and B can be used in scientific-engineering applications where a relatively small number of files are to be sorted. The trade-off between the desired degree of sorting efficiency and cost of the features for sharing the MM and/or the peripheral file should be decided at the user's level.

EFFICIENCY AND PERFORMANCE ANALYSIS

The efficiency of the SP depends upon the applications orientation, the size and the basic functional characteristics of the computer system. It is very difficult to depict generalized analytical expressions that correlate computer system parameters and sorting because of the diversity and inconsistency of the numerous variables involved.

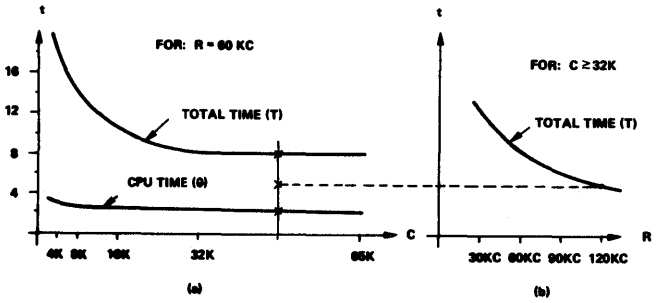


Figure 5—Statistical curves of sorting parameters

The diagrams of Figure 5 illustrate the correlation between the main sorting parameters: the sorting time (in minutes), the main memory capacity C (in kilocharacters), and the transfer rate R of the peripheral file (in kilocharacters per second).

These diagrams are derived by analyzing and combining the statistical data for two typical models of computer systems performing a sort.[3, 6, 7, 8]

The following conclusions can be derived:

a. The CPU time (θ) spent for sorting generally does not depend upon the size of the MM.
b. The increase of the transfer rate R causes a decline of the total sorting time T hyperbolically. The θ remains unchanged.
c. For $R = 60$ KC, the ratio between the I/O time ($T - \theta$) and θ equals 75 percent to 25 percent. This ratio is equal to 60 percent to 40 percent for $R \geq 120$ KC. This is the prevailing range of the transfer rates for the present magnetic files used in the small-to-medium and larger computer systems.

Considering the fact that at least 25 percent of the computer's workload in a business oriented system involves sorting, and 40 percent of that workload is the burden of the CPU, it is evident that the Sort Processor can release up to 10 percent of the CPU's overall working time.

The modular logic structure of the SP is highly adaptable to the further advances in LSI technology. Larger, faster and cheaper LSI chips (MOS or bipolar) can be easily utilized in the SP, improving the cost-performance index and increasing its overall efficiency in the computer systems.

The estimates show that the SP, designed with

today's off-the-shelf MOS LSI components, can save considerable amounts of user money.

SUMMARY

The semiconductor technology presently offers LSI components (specifically MOS memory chips) that have a very attractive price-performance index (less than ten cents per bit and around 100 nanoseconds access time). During the coming years this index will be subject to continuous and dramatic improvement thus setting up broader technical and economical grounds for hardware-software trade-offs. The purpose of these trade-offs is the simplification of the software sector of computer systems and the increase of the overall systems productivity for the user.

The Sort Processor designed with LSI components relieves the CPU from the burden of performing the tedious and time consuming sorting operations. It behaves like a low priority peripheral processor and does not cause any structural changes in the architecture of the computer system. For the small-to-medium and larger computer systems the Sort Processor can release up to 10 percent of the CPU's total workload. The techniques of the search memory and dynamic microprogramming allow use of the Sort Processor for algorithmic functions other than sorting.

ACKNOWLEDGEMENT

The author expresses his appreciation to the NCR–ED Research Department for encouraging the work on this project, and gratitudes to his colleagues: to Mr. A. G. Hanlon for stimulating discussions and advices, to Mr. D. W. Rork for his constructive engineering work in designing the breadboard of the Sort Processor, and to Mr. F. Sherwood for early discussions on sorting algorithms.

Special thanks are due to Mrs. Ann Peralta who performed the tedious job of typing and retyping this paper.

REFERENCES

1 The Diebold Research Program
Technology Series September 1968
2 I FLORES
Computer sorting
Prentice-Hall Inc 1969
3 *Computer characteristics digest*
Auerbach April 1969
4 D A BELL
The principles of sorting
Computer Journal Vol 1 No 2 June 1958
5 R R SEEBER
Associative self-sorting memory
Proceedings of EJCC Vol 18 pp 179–187 1960
6 *IBM system/360 disc and tape operating system. Sort/merge program specification*
File No S360–33 Form C24/3444
7 *The National 315 electronic data processing system sorting tables, magnetic tapes*
The National Cash Register Company Dayton Ohio
8 *Magnetic tape sort generator*
Reference manual The National Cash Register Company
Dayton Ohio

System/360 model 85 microdiagnostics

by NEIL BARTOW and ROBERT MCGUIRE

International Business Machines Corporation
Kingston, New York

INTRODUCTION

System/360 Model 85 is a large central processing unit, (CPU), which contains a machine cycle of 80 nanoseconds and a main storage access of 1.2 microseconds. It has the capability of executing 12,500,000 add register to register type instructions per second. Its major parts are the Instruction Preparation Unit, (I Unit), Instruction Execution Unit, (E Unit), and Storage Control Unit, (SCU). In addition to these three main parts, there is also another portion of the hardware dedicated to maintenance controls. The IBM Model 85 computer has high speed buffer storage and hardware capable of initiating and executing Instruction Retry. There are two major control storage elements. Read Only Storage, (ROS), and Writeable Control Storage, (WCS). ROS consists of 2,048 decimal control words while WCS consists of 512 control words for a standard Model 85 or 1024 control words for a Model 85 with an emulator feature. Each control word is 128 bits long and consists of 33 control fields as illustrated in Figure 1—Model 85 Control Word. Approximately 450 microorders have been implemented in the Model 85 for use in microprogramming.

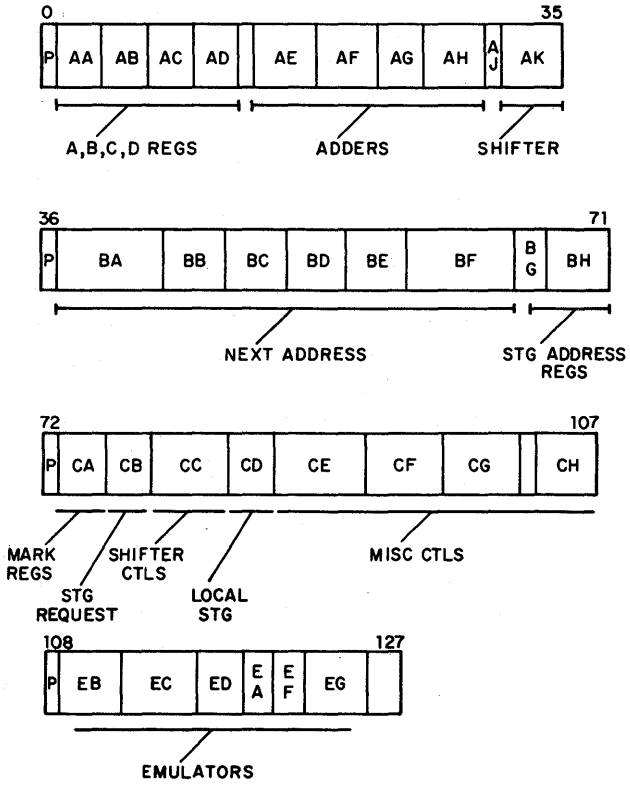


Figure 1—Model 85 control word

DEFINITIONS

Microprogram—a computer based program whose microinstruction set is geared to one or more logical hardware functions which are executable in 'one machine cycle.' Two or more microinstructions are normally required for the execution of one instruction of the standard instruction set.

Microdiagnostic—a microprogram designed specifically to test a predefined portion of hardware.

Microdiagnostic Example

Figure 2—Microdiagnostic Test (STAT 'A')—is an illustration of a microdiagnostic test. The test uses the Control Automated System, (CAS), output for its descriptive representation. Each block of the test, of which there are four, represents one machine cycle. The purpose of the test is to confirm that the hardware required to set a latch called STAT 'A' is working properly. If it is not, to stop at control storage address A02.

Cycle 1 is defined by the control word located at control storage address A12 (Hex). This control word will reset a latch called STAT 'A' and fetch the control

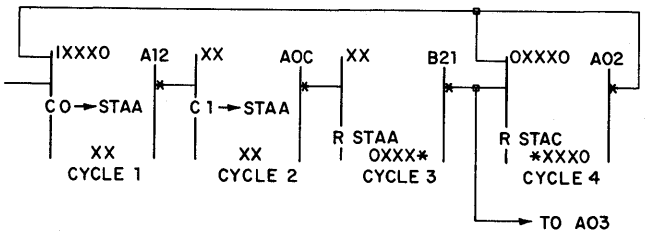


Figure 2—Microdiagnostic test (STAT 'A')

word located in control storage address A0C to the control registers. Cycle 2 is defined by this control word at control storage address A0C. This control word will set the latch called STAT 'A' and cause the control word at control storage address B21 to be fetched to the control register. Cycle 3 is defined by the control word at address B21 which will set the 12th bit of the next address field to 1 making the next address A03 if a latch called STAT 'A' is in the set state. Or, the 12th bit of the next address field will remain 0 if the latch called STAT 'A' is in the reset state. Assuming STAT 'A' is reset the control word at control storage address A02 will be fetched to the control register. Cycle 4 is defined by the control word A02 and will set the 8th bit of the next address field to 1 if the Latch call STAT 'C' is in the set state making the next address A12. Or, the 8th bit of the next address field will remain 0 if the latch called STAT 'C' is in the reset state causing the next address to be A02.

STAT 'C' can be set or reset by a toggle switch labeled 'Loop Test' on the maintenance console. Since diagnostics are run with the loop test switch in the off position this test will stop at address A02 if STAT 'A' fails to set and the test can be looped simply by setting of the loop test switch.

MICROPROGRAMMING USAGE

Base machine functions

The System/360 Model 85's basic machine functions are defined by control words contained in Read Only Storage. They consist of control words for machine instruction execution and sequencing of manual control functions which are initiated from the maintenance console. Other functions include control words for the retry of failing instructions, interrupt sequencing and provisions for handling of invalid instruction operation (op) codes.

Loading of WCS

Load WCS routine

WCS loading for microdiagnostics is handled from a routine in ROS and is designed to load 512 control words from main storage into WCS. This routine can be executed from one of two entry points—either by using the address contained in the double word starting with main storage address 8 or by establishing a value in one of the internal working registers and bypassing that part of the routine which fetches main storage address 8.

LMP instruction

WCS loading for purposes other than microdiagnostics is normally handled by the Load Microprogram Instruction. This instruction is a privileged member of the System/360 instruction set. It is capable of loading one to four control words into WCS from main storage. The control words are indexed by the operand field of the instruction.

Emulators

The IBM 7090/7094 Emulator is a prime application for System/360 Model 85 microprogramming. When emulators are installed on the machine it is necessary to modify the instruction preparation unit in order to handle the additional operation codes required for emulator instruction. WCS must be expanded to two times its basic size, that is, 1,024 decimal control words, and must be loaded with the control words which are required for the execution of each emulator instruction.

Multiply Algorithm

The low speed multiply algorithm contained in WCS is an alternate way of executing the multiply instruction when the high speed multiply feature is installed on the machine. The high speed multiply feature requires its own dedicated circuitry in the E unit. The low speed multiply algorithm is used when there is a failure or malfunction in the high speed multiply hardware. The low speed multiply algorithm is activated by setting of a system mode latch which is done normally via the Diagnose instruction.

Hybrid diagnostics

Hybrid diagnostics are another form of microprogram usage with the System/360 Model 85 as shown in

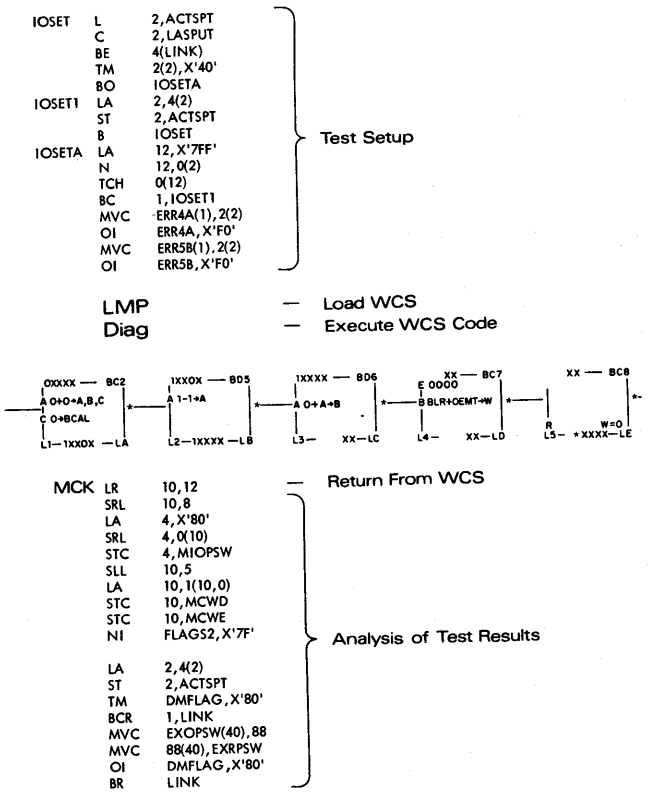


Figure 3—Hybrid diagnostics

Figure 3—Hybrid Diagnostics. The first set of instructions (SET UP) is conventional System/360 code which is used for all initialization required prior to executing the test.

The next step is the Load Micro Program, (LMP), instruction which loads the test control words into Writable Control Storage. The Diagnose instruction indicated by DIAG is a privileged instruction of the System/360 instruction set. It is used to branch from System/360 code into WCS and turn control over to those control words which have just been loaded via the LMP instruction. These control words, illustrated by the CAS diagram in Figure 3 are executed in sequence. At the end of the sequence, a pseudo machine check takes control from WCS and returns the program to the normal System/360 mode at the point indicated by RETURN FROM WCS. Conventional code is then used for an analysis of test results. This is one way of writing the hybrid diagnostic test.

The control words used for this test could just as well have been used for the test setup or they could have been used to analyze test results, with conventional code used for the other two parts of the diagnostic.

Microdiagnostics

Microdiagnostics are microprograms specifically designed to test a given hardware function. Their main purpose is to detect basic machine malfunctions and to isolate the failing components. There are three parts to microdiagnostics:

- First, the resident diagnostics which are located in ROS
- Second, the non-resident diagnostics which are found in WCS.
- Third, the loader which is used to bring non-resident diagnostics into main storage. The loader is also executed from WCS.

Resident microdiagnostics

Resident microdiagnostics are used to test all the data paths and microorders needed to execute the Initial Program Load, (IPL), and the Load WCS routine.

Non-resident microdiagnostics

Non-resident microdiagnostics are used to test the remainder of the basic machine functions. They start testing the E unit and progress to the I unit and then the SCU. At the end of execution of these diagnostics, control is turned over the the diagnostic monitor and and conventional System/360 diagnostics are executed. Non-resident microdiagnostics can be loaded into WCS from card, tape or disk I/O devices.

There are approximately 30 sections of non-resident microdiagnostics. Each section contains a maximum of 512 control words.

Microdiagnostic Loader

The microdiagnostic loader is part of the first non-resident microdiagnostic program found on the diagnostic I/O device. It is used to load into main storage the non-resident microdiagnostic programs during the execution of the entire microdiagnostic package. The

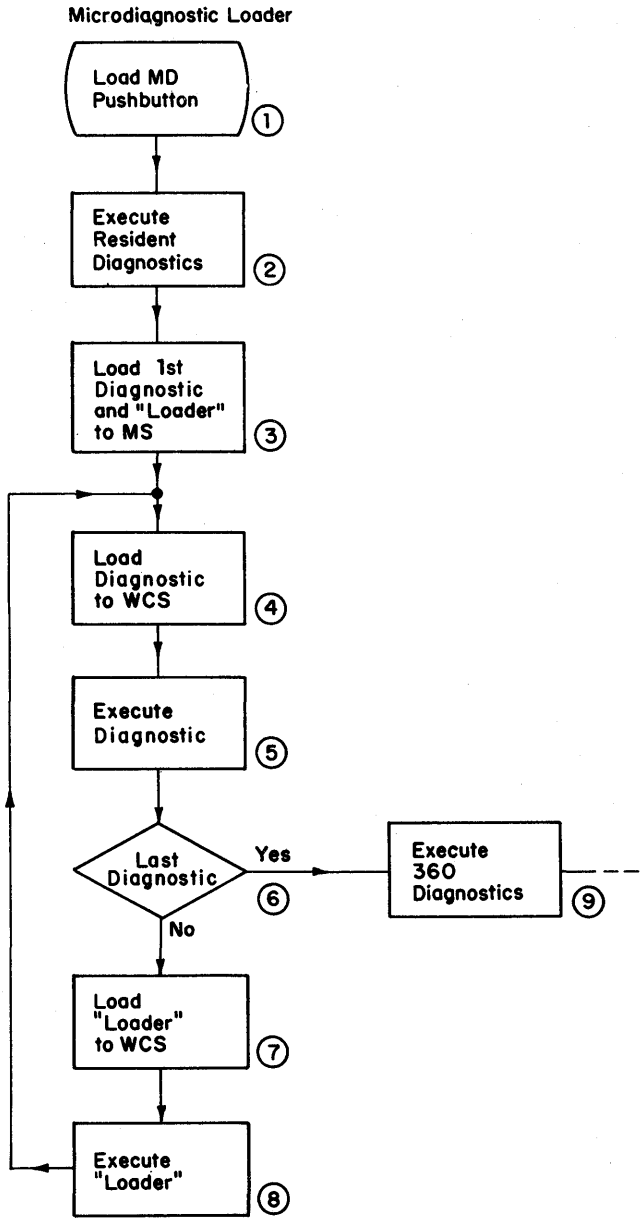


Figure 4—Load sequence flowchart

loading sequence is illustrated in Figure 4—Load Sequence Flowchart.

The flowchart shows microdiagnostics are initiated with the depression of the Load Microdiagnostic Pushbutton (LMD), found on the maintenance console at Reference 1. The depression of this pushbutton causes a system reset and control to be given to the resident microdiagnostics in ROS. Resident microdiagnostics are then executed at Reference 2. If the resident microdiagnostics are successful, a modified Initial Program Load, (IPL), will cause the first non-resident microdiagnostic program to be loaded into main storage from the I/O device at Reference 3. This microdiagnostic contains the first microdiagnostic and the microdiagnostic loader. At the end of the modified IPL, the first non-resident microdiagnostic is loaded into WCS by the 'LOAD WCS ROUTINE' in ROS at Reference 4. This diagnostic is then executed at Reference 5. If it is successful, control is given to the 'LOAD WCS ROUTINE' and the loader is brought into WCS, overlaying the first microdiagnostic at Reference 7. The loader is then executed at Reference 8. This causes the second non-resident microdiagnostic to be brought into main storage from the I/O device. The loader then gives control to the 'LOAD WCS ROUTINE', Reference 4, which will load the second non-resident microdiagnostic into WCS. This microdiagnostic is then executed at Reference 5 and if it is not the last microdiagnostic, Reference 6, it will give control to the 'LOAD WCS ROUTINE' again, and bring the loader into WCS at Reference 7. If it is the last non-resident microdiagnostic at Reference 6 it will start the I unit and give control to the conventional System/360 diagnostics at Reference 9.

Using this loading technique, resident microdiagnostics, non-resident microdiagnostics and conventional System/360 diagnostics can be executed in sequence without manual intervention simply by depressing the Load Microdiagnostic pushbutton.

It should be noted that the time in which all of the non-resident microdiagnostics are executed is approximately 20 seconds. It should also be noted that conventional System/360 diagnostics which follow the non-resident microdiagnostics are by no means redundant tests. The conventional diagnostics provide functional tests which have not been attempted in the non-resident and resident microdiagnostics. In addition to this, they provide systems tests and tests of channel and control unit hardware, again, not attempted in the microdiagnostics.

ADVANTAGES OF MICRODIAGNOSTICS

Start small philosophy

Of prime importance in microdiagnostics is a philosophy called "START SMALL". The "START SMALL" philosophy is a building block approach to diagnostics which uses an assumption of a solid hardware failure. The object is to establish a known portion of the hardware to be operating properly so that it may be used to check additional hardware whose condition is unknown. Using this technique it is only necessary to compare one test result against one result generated by tested hardware.

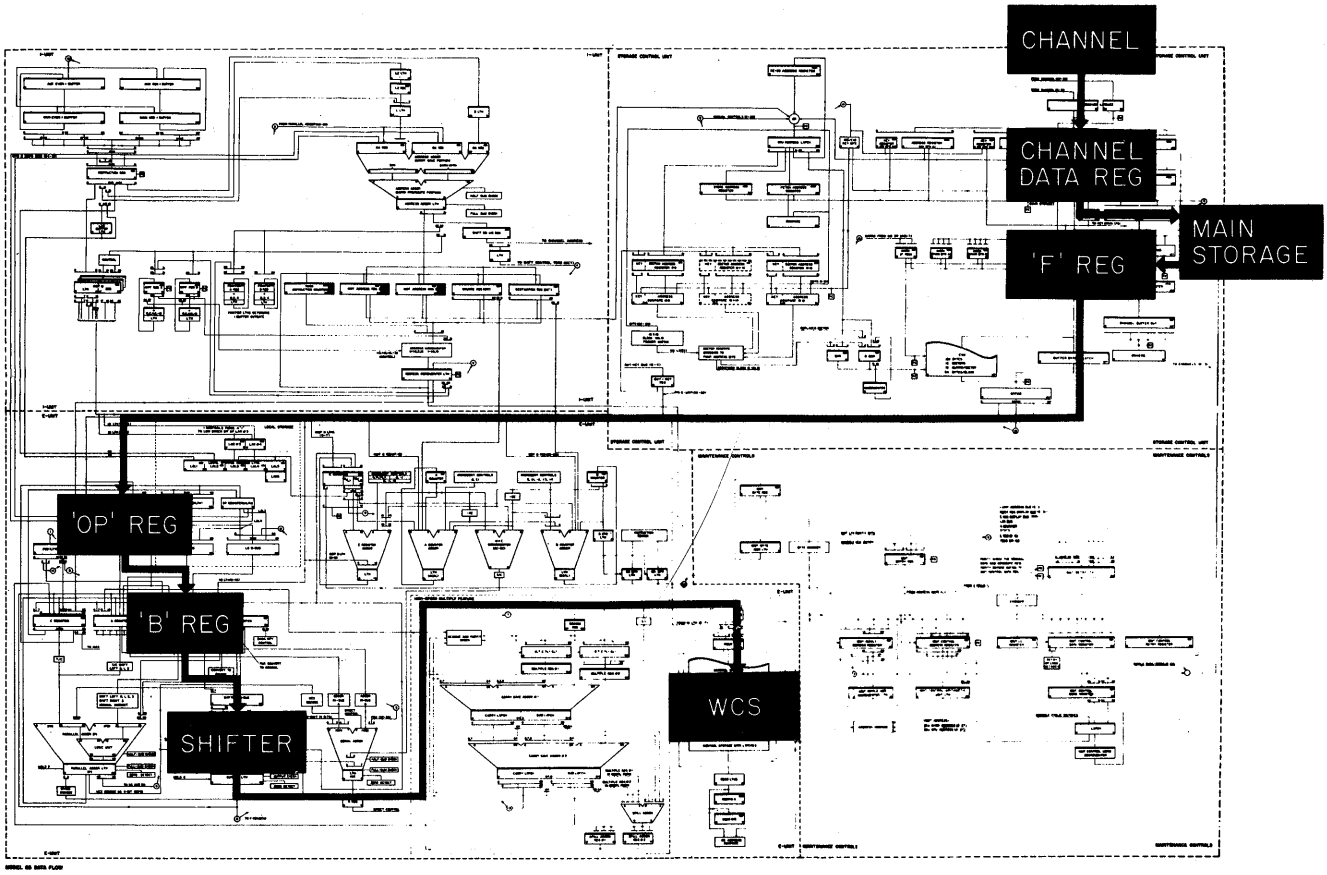


Figure 5—Load path data flow

Initial hardware required

The "START SMALL" philosophy as implemented under conventional diagnostics, requires that a large part of the storage control unit and I unit and a significant portion of the E unit be working before an instruction of the System/360 set can be executed. In comparison, the hardware required for resident microdiagnostics to be executed is only a small portion of the E unit. This is why a "START SMALL" philosophy as implemented with microdiagnostics is considerably more effective than the technique implemented with conventional diagnostics.

Load path

The second advantage of microdiagnostics is the ability to check out the load path from the I/O channel to WCS. This path, shown in Figure 5—Load Path Data Flow—is from the channel to the channel data register and into main storage. Data is moved from main storage through the F register into the E unit operation register, then to the B register to the shifter and WCS. Conventional diagnostics require that the data path from the channel to main storage be working before data can be loaded into memory prior to execution. In the case of microdiagnostics, the residents have the ability to check the data path from the I/O device all the way to WCS prior to executing and I/O operation. This procedure insures that the path is operational and that non-resident microdiagnostics can be loaded successfully. In addition to checking the load path, the resident microdiagnostics have the ability to disable the I unit and the high speed storage buffer in the SCU. This procedure avoids certain portions of the circuitry which would be used during a normal IPL sequence. As a result, the possibility of errors occurring from that circuitry not essential to load data from an I/O device into WCS is reduced.

Increased isolation

Microdiagnostics provide for increased isolation in the I unit, E Unit and SCU by taking advantage of the fine control which they can exert on the CPU hardware.

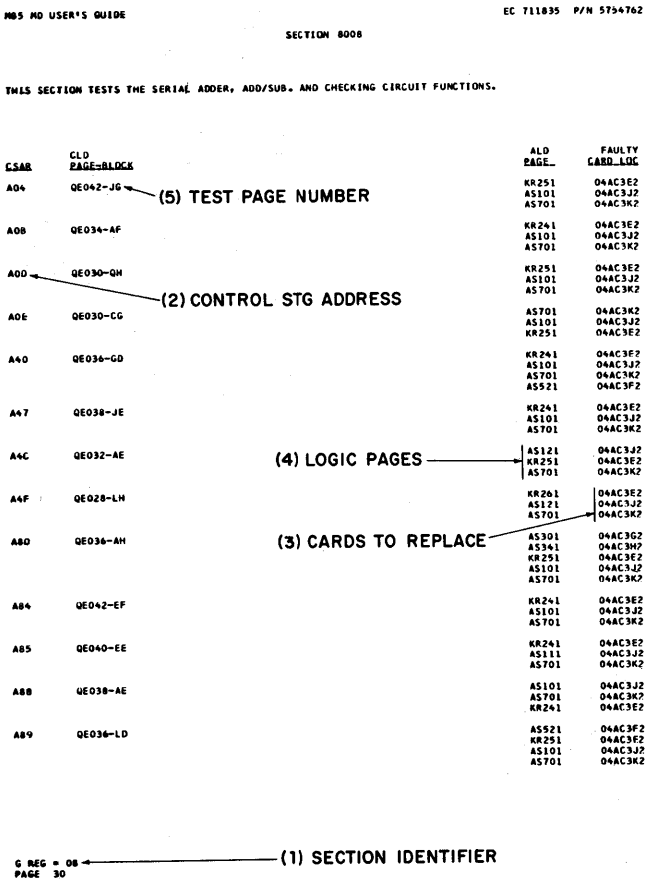


Figure 6—Microdiagnostic cross reference list

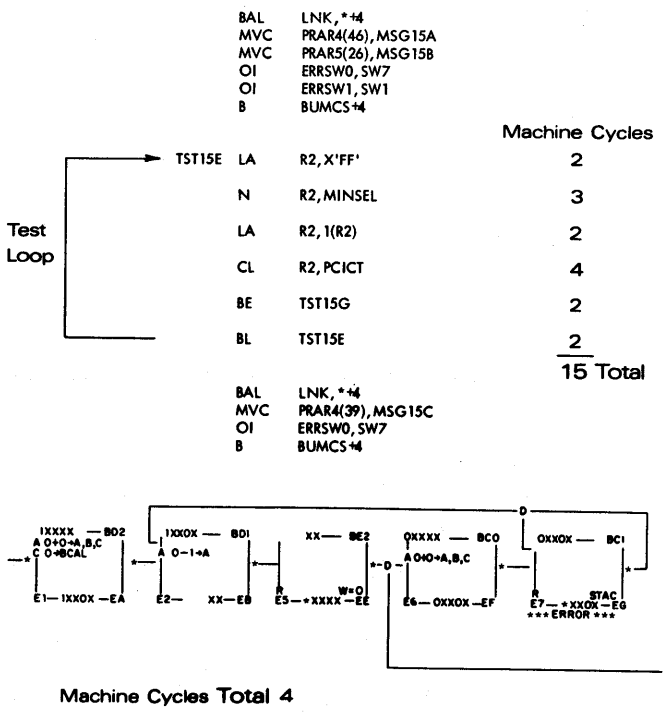


Figure 7—Tight scope loop

In order to eliminate the necessity of using untested hardware only the simplest of registers are used for CE communications. In conjunction with this technique, the Model 85 has implemented a cross reference list (CRL), as shown in Figure 6—Microdiagnostic Cross Reference List—that uses a register, Reference 1, and a control word address Reference 2. This information is displayed on the maintenance console to direct the Customer Engineer, (CE), to a list of two to ten Field Replaceable Units, (FRU), for any one error point, Reference 3. In addition, the cross reference lists include logic pages for hardware being tested, Reference 4, as well as a flow diagram page number of the test at Reference 5.

Tight scoping loop

Scoping loops are program loops used by Customer Engineers to scope failure areas in a machine. The Customer Engineer has the option of implementing a scoping loop in the event the cross reference list does not indicate the correct Field Replaceable Unit (FRU). These loops are considerably shorter on the System/360 Model 85 microdiagnostics than in conventional diagnostics. The upper portion of Figure 7—Tight Scope Loop—indicates 15 machine cycles for a scoping loop in conventional diagnostics as opposed to four machine cycles required for a loop in microdiagnostics found in the lower portion of Figure 7. The total fifteen cycles for conventional diagnostics is highly conservative since the instructions chosen to illustrate the loop are short and require few machine cycles. The microdiagnostic loop, however, is quite typical of those tests implemented in System/360 Model 85. The maximum number of machine cycles that would be found in the microdiagnostic loop is in the order of ten. In the case of conventional diagnostics, the maximum number of machine cycles could be in the hundreds.

Manual controls

Special microorders implemented in the System/360 Model 85 have made manual control over the microdiagnostics very effective. All tests implement a branch of a maintenance control switch which loops a diagnostic test to provide a scoping loop for the Customer

Engineer. In addition, all sections can be looped via another switch on the console. The implementation of microorders to sense switches is simple in comparison to the technique used in conventional diagnostics. Switches on the console are also labeled as an aid for the CE. Conventional diagnostics on the other hand, require special instructions in order to sense switches on the console. The sensing of these switches is rather lengthy in terms of execution time, therefore, it is not normally done during scoping loops.

ROS simulation

Writeable Control Storage in System/360 Model 85 has been used to test resident diagnostics implemented in Read Only Storage (ROS). This procedure has proved to be a very effective and efficient way of implementing code found in Read Only Storage because turnaround time for manufacture of bit planes necessary for ROS is relatively long in comparison to that for assemblies used for WCS. In addition to the improvement in speed and flexibility, significant cost savings have been realized by eliminating unnecessary manufacture of incorrect bit planes. Since WCS can be expanded to the same size as Read Only Storage it can be used to check the basic machine set as well as the resident microdiagnostics.

LIMITATIONS OF MICRODIAGNOSTICS

Limitations in implementing System/360 Model 85 microdiagnostics fall into one of five main categories:

- First - it is impractical to implement functional testing in microcode. Functional tests imply that the functional design of the machine is tested. If we substitute WCS code for the functional ROS code, we would not be testing the design of the machine.
- Second - limited access is available to the I unit and the Storage Control Unit. This is understandable because microcoding was used in the design of the System/360 Model 85 primarily for the control of the execution unit. The I unit and the SCU are primarily controlled by hardware sequencing.
- Third - coding inefficiencies for large data handling are prevalent because of the lack of power of microinstructions.
- Fourth - interactive problems (timing) require a considerable amount of test setup which is more efficiently done in conventional code. In addition, the interactive problems of prime concern are those which exist between instructions of the basic instruction set (functional).
- Fifth - skill level - because microprogramming is such a low level language, detailed knowledge of the hardware is required before any attempt can be made to program the machine.

SOFTWARE SUPPORT OF MICRODIAGNOSTICS

The system which was initially adopted for System/360 Model 85 support is the Controlled Automated System (CAS). This system yields a flow chart of microinstructions being executed. It is designed for use by engineers for implementation of their code in Read Only Storage and has proved to be an asset in development of design. This system, however, does not provide flexibility needed for the programming environment of microdiagnostics. Because of this lack of flexibility an interim assembler was developed for the support of the microdiagnostics debug. This program provides a listing format. Utilities are available to permit changes in program decks and a loader which can be used to load these decks into WCS for debugging purposes.

Use of read only memory in ILLIAC IV*

by H. J. WHITE and E. K. C. YU

Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

Because of its high speed operation, large instruction repertoire and centralized control, the ILLIAC IV Computer uses a Read Only Memory (ROM) to translate instructions into control enables. These control signals are broadcast to the array of parallel processors to control the step by step operation of each processor. Each of the over 260 instructions is decoded into a microsequence (microprogram) used to address the ROM. Each microsequence consists of from one to 69 microsteps (microinstructions).

The ROM is a transistor cross point matrix and is configured with discrete transistors mounted on large multilayer circuit boards. The memory size is 720 words (microsteps) by 280 bits (control enables). Cycle time is 50 nsec.

To simplify instruction decoding, up to five words are simultaneously addressed in many microsequences, i.e., control enable ORing. This ORing also improves memory speed and reliability by greatly reducing the number of transistors required. To save execution time, up to two instructions are simultaneously addressed, i.e., instruction overlap. Thus, up to ten memory words could be simultaneously addressed. Because the Read Only Memory uses linear addressing, the simultaneous addressing of any number of words is easily achieved.

THE ILLIAC IV SYSTEM

By using extensive parallel processing, the ILLIAC IV System (1) offers computing power capable of solving a number of problems beyond the power of currently available or proposed computers. Some problems involve manipulations of very large matrices (e.g., linear programimng), others involve the solution of sets of partial differential equations over large grids (e.g. numerical weather prediction); and still others require extremely fast correlation techniques (e.g., phased array radar).

As shown in Figure 1, the IILLAC IV System consists of a large parallel-array computer coupled to an I/Ø subsystem. The I/Ø section contains

(1) A Burroughs B6500 Computer which functions as the executive element.

(2) Two Burroughs disk files. (Each file is capable of up to 16 parallel disks. Each disk contains about 79×10^6 bits of storage. The effective bit transfer rate of each disk file is over 500×10^6 bits per second. Both files may operate concurrently for a net maximum transfer rate of 10^9 bits per second.

(3) An I/Ø Controller, buffer memory and switch for interfacing with the computer.

The parallel array computer consists of four identical array processors or quadrants. Figure 2 illustrates the quadrant diagram. Each quadrant consists of a Control Unit (CU) and 64 Processing Units (PU's). Each PU, in turn, contains a 10,000 gate arithmetic unit called the Processing Element (PE) (2), a 2048 word by 64 bit semiconductor memory called the Processing Element Memory (PEM) and a 1000 gate interface unit called the Memory Logic Unit. All the controls normally associated with an arithmetic unit are extracted from the PE's and placed in the CU. These controls are shared in parallel, by all 64 PE's in the quadrant.

Both PE data and CU instructions are contained in the array of 64 PEM's which serves as the main memory for the quadrant. The CU has access to the entire PEM array, while each PE can only directly reference its own PEM.

The following data give some idea of the size and complexity of the ILLIAC IV Computer (not including the I/Ø subsystem). Each quadrant is 53 feet long, 6.5 feet deep and 8 feet high. Each quadrant contains about

* Project supported by Advanced Research Projects Agency under Contract No. AF30(602)4144 as administered by University of Illinois, Urbana, Illinois.

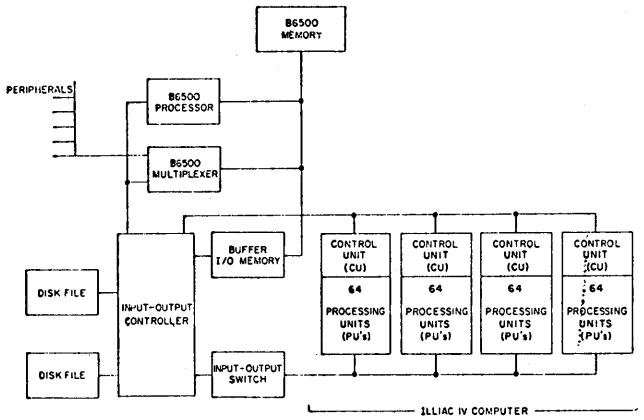


Figure 1

'850K, specially developed, high speed, emitter-coupled-logic (ECL) gates in a total of about 240K dual-in-line packages. Each quadrant requires about 210 KW of input power. Each quadrant contains 128 K words of storage at 64 bits per word. Memory access time is 188 nsec and cycle time is 200 nsec. The entire system operates at a 20 mHz clock rate. It performs a 64 bit floating point add in 250 nsec, a 64 bit floating point multiply in 450 nsec and a 64 bit floating point divide in 2800 nsec.

INSTRUCTION FLOW THROUGH THE CONTROL UNIT

The primary functions of the CU are:

(1) Control and decode instruction streams.

(2) Generate the control signals transmitted to the processing elements for instruction execution.

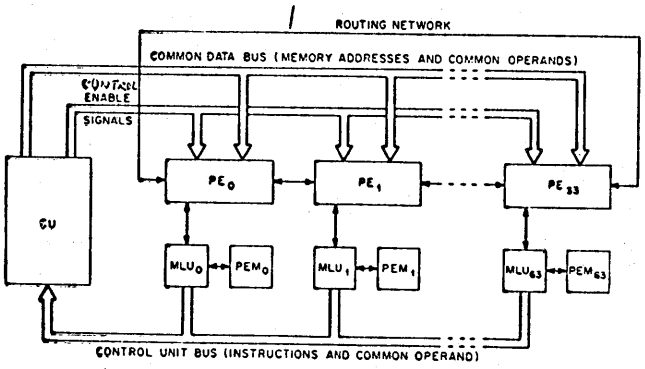


Figure 2

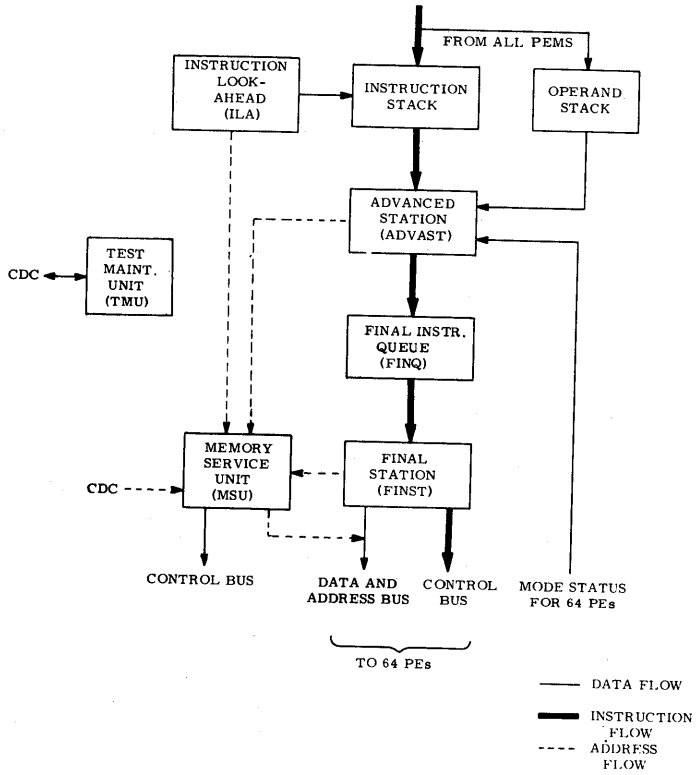


Figure 3

(3) Generate and broadcast data words and those components of memory address that are common to all PE's.

(4) Receive and process traps signals arising from arithmetic faults in the PE's or operations in the I/O subsystem.

The major parts of the CU, as shown in Figure 3, are:

(1) ILA—Instruction Look Ahead, whose function is to fetch and store large blocks of contiguous code. It fetches 8 word blocks having two instructions per word. It stores up to 8 blocks, i.e., 128 instructions.

(2) ADVAST—Advanced Station, is the principal housekeeper of the system wherein such functions as address arithmetic, loop control, interrupt processing and configuration control are performed. It receives and processes instructions from ILA.

(3) FINST—Final Station has the function of directly controlling the operation of the PE's in the array.

(4) MSU—Memory Service Unit resolves conflicts among all users requesting access to the array memory.

(5) TMU—Test and Maintenance Unit.

From a programming standpoint, FINST and the PE's perform "inner-loops" of a program and ADVAST performs "outer-loops" and control functions. When ADVAST receives a PE instruction, it performs the necessary indexing (if required) and then sends it directly to FINST for processing. Instructions enter FINST thru a first in-first out queue (FINQ) which contains up to 8 instructions. The queue decouples FINST execution time from ADVAST execution time and permits instruction overlap between the two stations. Any synchronism during overlap is handled by hardware.

Using a Read Only Memory (ROM), FINST decodes instructions into control enables which are then broadcast to the array of 64 PE's. The control enables control information flow both in direction (register to register) and time. The ROM is a transistor crosspoint matrix.

In FINST two instructions are examined concurrently for purposes of instruction overlap, e.g., to see if data fetches for the next instruction can take place during the end of the present instruction. Instructions first appear in an overlap section from which preliminary commands are generated and then appear in an instruction section in which the remainder of the instruction is completed. For many instructions, two or more words are simultaneously addressed, i.e., control enable ORing. This ORing simplifies branching within the instruction and also improves memory speed and reliability by greatly reducing the number of transistors required. Thus, up to ten words are simultaneously addressed.

REASONS FOR USING A READ ONLY MEMORY

The ILLIAC IV instruction repertoire consists of over 260 instructions. This large number of instructions provides for great system flexibility.

For example, there are six kinds of signed and unsigned arithmetic:

1. Normalized floating point.
2. Normalized floating point rounded.
3. Unnormalized floating point.
4. Unnormalized floating point rounded.
5. Mantissa-sized fixed point.
6. Mantissa-sized fixed point rounded.

All data operations can be performed in 64-bit or 32-bit mode. Also, there is a limited amount of unsigned full word (64-bit) and 8-bit arithmetic. Thus, the system can be operated in an 8-, 32- or 64-bit mode which, for a 256 PE system, gives the capability of 512 32-bit processors or 2048 8-bit processors.

In FINST, the 260 instructions are decoded into 280 control enables of which about 260 are broadcast to the PE's with the remaining enables controlling areas in the CU. The reasons for the large number of PE enables are:

1. The large variety of arithmetic modes.
2. Centralizing all processing controls in the Control Unit.
3. High speed operation, i.e., control enables are changed every clock cycle (50 nsec).

Thus, the primary functions of FINST is to decode a large number of instructions (260) into a large number of enables (280). Two methods of decoding instructions were investigated: (1) a fully hard wired logic approach and (2) the use of an ROM. In the logic approach, all instructions generating the same control enable are essentially OR'd together. To meet the system speed requirements, all instructions generating the same control enable, should be on the same multilayer P.C, board. This leads to considerable duplication because the number of instructions generating a control enable may be so large that only a few of the complete set o. enables can be on the same board. In this approachf each board had an identical instruction register, timing counter and other basic logic required to decode instructions. A minimum of 60 boards was estimated to be required and the most likely number was about 80. Each board was unique which presented a serious board sparing problem. In the ROM approach, which was adopted, a total of 28 boards is required. There are eight word driver boards which generate the microsequences required to address the memory. Each of these boards is unique. There are 12 matrix boards which make up the memory. Each matrix board is identical except for the location of cross point transistors. There are eight sense amplifier boards. Although only one type of board is required, two types of boards were used as a convenience. Thus, the ROM approach occupies less than half the volume, had fewer board types and uses less than about 1/4 the number of components. In addition, the logical design is simpler in the ROM approach and changes are more easily implemented. Although an MSI or LSI ROM might be cheaper and faster, a discrete component ROM design was adopted for the following reasons:

1. A suitable, I.C., ROM would not be available in time (by first quarter of 1970).
2. Changes are easily made at any time in a discrete memory.
3. Linear addressing is easily accomplished in a discrete memory. Linear addressing, i.e., one address line per word, simplifies the simultaneous addressing

required to achieve instruction overlap and control enable ORing.

INSTRUCTION OVERLAP

As stated earlier up to ten ROM words are simultaneously addressed as PE instructions are processed in FINST. For any instruction, up to five words are addressed to achieve control enable ORing. In addition, to realize the time saving of instruction overlap, up to two instructions are decoded at the same time.

Figure 4, illustrates how instruction overlap is accomplished. As instructions are received from ADVAST they are stacked in an eight word queue, FINQ. When an instruction is in position 1, it is transferred to the FINST Overlap Register (FOR). In FOR two actions take place. First, the instruction is examined to determine what parts of the Processing Elements will be used when the instruction is transferred to the FINST Instruction Register (FIR). This information is stored in the 1st level of a Busy Register. The Busy Register also contains the same PE usage information, in level 2, for the next intsruction to be executed, which is in FIR, and contains like information, in level 3, for the instruction presently being executed, which is in FIAR (the ROM address register associated with FIR). The instruction in FOR is also examined to see if it is a candidate for instruction overlap. If it is, it is decoded into a microsequence which addresses the overlap section of the ROM (250 addresses) using the ROM address register FOAR. If the required sections of the PE are busy, the microsequence is inhibited until the PE sections are free.

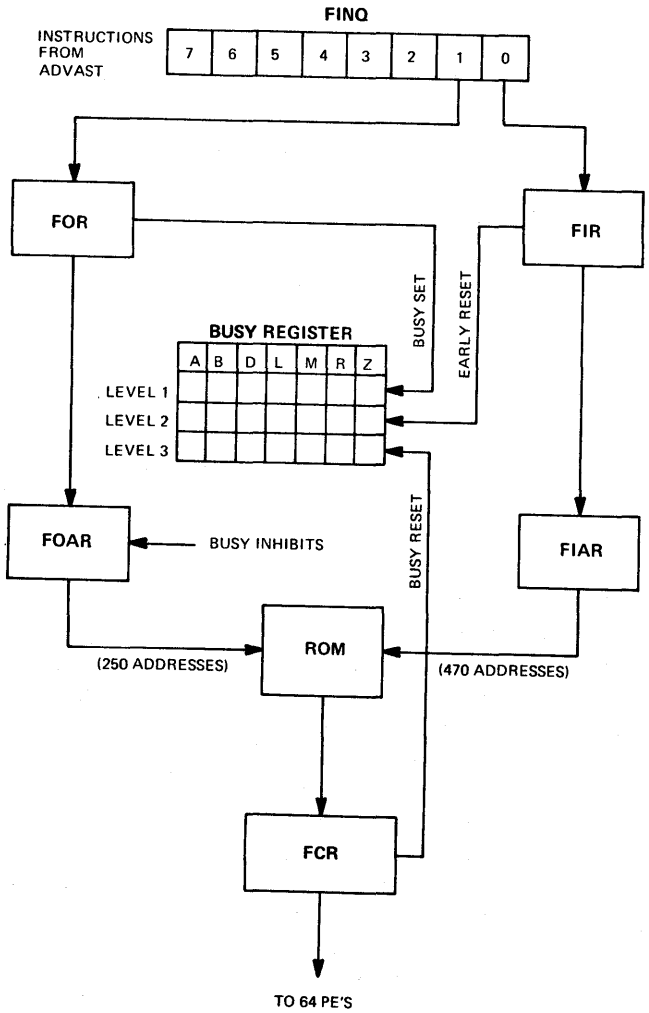


Figure 4

FIAR addresses a 470 word section of the ROM which is sufficient to execute all instructions. When the instruction in FIAR is fully executed, the instruction in FIR is transferred to FIAR and the instruction pointers to the queue are incremented one position. This shifts the instruction from FOR to FIR and puts a new instruction in FOR. At the same time, the Busy Register is updated to determine if instruction overlap is possible.

To achieve the required high speed operation, the PE busy bits in word one are set while the instruction is in FOR. As the instruction moves from FOR to FIR to FIAR, the busy bits are transferred from level 1 to level 2 to level 3. Busy bits are reset via two paths. The normal path is via CU control enables out of the ROM. This path takes the longest (3 clock times) when the instruction in FIR is transferred to FIAR. Because this may delay the start of an instruction overlap, early resets are generated in FIR and enabled when the instruction transfers to FIAR.

INSTRUCTION DECODING

Instruction decoding is the same whether it occurs in the overlap or instruction stations. The format of the twelve bit instruction word is shown in Figure 5(a). Bit 0 indicates whether the operation is in 32-bit or 64-bit mode. Bits 9, 10 and 11 are operations on the data word associated with the instruction such as address indexing when the data word is an address. Bits 1 thru 8 contain the OP CODE. Each instruction is decoded into a microsequence (microprogram) used to address the ROM. Each microsequence consists of from one to 69 microsteps (microinstructions). Generally each microstep is an ROM word. In some operations, such as divide, the same word is addressed many times in succession. However, each time the word is addressed it is considered a microstep. Since a word is addressed

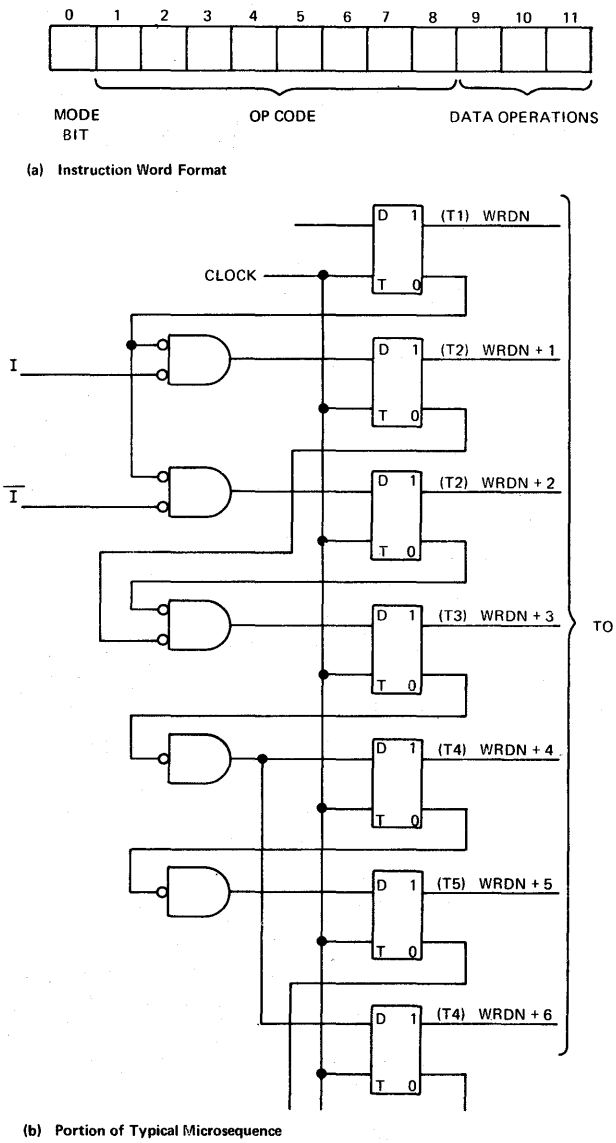


Figure 5

every clock cycle, microsteps are synonymous with clock times. When many words are addressed simultaneously to achieve control enable ORing, this is also considered a single microstep. Each microstep generates a full set of control enables which are stored in the FINST Control Register (FCR). From FCR they are broadcast to the 64 PE's in the quadrant (see Figure 4). Generally from one to 50 enables are active for each microstep.

The decoded instruction contains the starting address of the microsequence and all information for decision making, such as branching, within the microsequence. Typical branches are one of the six ways to do signed and unsigned arithmetic operations. Figure 5b illustrates a portion of a typical microsequence. The seven flip flops shown are "D" type flip flops, i.e., the "1" output is in the same state after the clock pulse as the "D" input was during the clock pulse. Each "1" output is designated by the ROM word it addresses. Related clock times are shown in ( ), e.g., (T2). The flip flops are part of the address register (FOAR or FIAR). Under control of the decoded instruction, the microsequence proceeds from flip flop to flip flop each clock time. Referring to Figure 5b, at time T1, word N is addressed. At time T2, either word N + 1 or N + 2 is addressed depending on the state of control bit I. At time T3, word N + 3 is addressed. At time T4 both words N + 4 and N + 6 are addressed to obtain enable ORing. Finally, word N + 5 is addressed at time T5.

The following microsteps are the microsequence for a floating point add and are accomplished in 250 ns (five clock times):

(1) Fetch Operand. Transfer to B register (RGB), in the PE, the operand identified by the address field of the instruction.

(2) Difference Exponent. Subtract exponent fields of operands in PE A register (RGA) and RGB.

(3) Mantissa Alignment. Shift mantissa of operand in RGA or RGB by amount determined from step 2.

(4) Add Mantissa. Add mantissa field of operands in RGA and RGB.

(5) Normalize. Normalize sum in RGA.

BASIC ROM REQUIREMENTS

A Read Only Memory (ROM) may be thought of as a numerical conversion table, i.e., the selection of one of the input lines will present a set of predetermined numbers at the output of the memory. It is a simple matter to design a conversion table to read-out one set of numbers corresponding to the selection of either one or more than one input numbers. In block diagram form, such an ROM is shown in Figure 6. However, in this paper, only

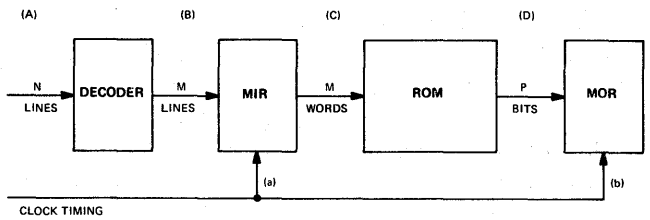


Figure 6

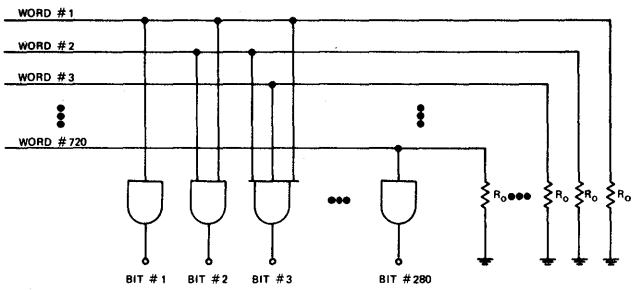


Figure 7

the functional block which performs the numerical conversion, is called the Read Only Memory. The other functional blocks are considered to be peripheral logical functions. The logic design of the ILLIAC IV calls for an ROM having a cycle time of 50 nanoseconds, i.e., the ability to read out a set of numbers every 50 nanoseconds. Referring to Figure 6, the timing from the input register (MIR) to the output register (MOR) is 50 nsec. The memory must accept 720 input lines and present an output of 280 bits. To be useful, an ROM must be alterable either electrically or mechanically. Note that an electrically alterable memory is essentially a read/write memory and is, as a rule, more difficult to construct than a memory that is altered by mechanical means.

Because of its large size and fast cycle time, only an ROM in a matrix form, was found satisfactory for use in ILLIAC IV. It may be noted that an ROM can be constructed with sufficient speed by employing ECL gates with typically 2.5 nanosecond propagation delays. Such a design is schematically shown in Figure 7. In principle, the ROM is simply P number of gates where each gate is one output bit of the memory, and M line drivers, corresponding to the M input words. The gates are selectively connected to the input word lines, such that the activation of any one or more word lines provides a predetermined output bit pattern through the gates. The design of Figure 7 was considered early in the ILLIAC IV development. But careful examination of the design reveals that there must be 720 input lines and a minimum of 280 output gates. Each output gate must have 720 inputs. For gates with only 9-inputs (maximum available), each of the output gates must be connected with 80 gates in parallel to accept the 720 possible input lines. The 80 gates produce only one output bit, and must be buffered with multiple stages of OR-gates to provide that one bit output. Similarly, multiple buffering or amplifying stages must be used to drive the large number of output gates. Because of the large number of gates involved, inter-connection wiring is complex. An almost insurmountable difficulty in such a design is to change the content of the memory, mechanically or otherwise. By using a matrix design, the number of printed circuit boards is reduced and the memory is readily altered.

ROM DESIGN DETAILS

The ROM schematic is shown in Figure 8. The form is a standard, transistor cross point matrix consisting of m (720) word lines by n (280) bit output lines. Only those bit lines that are transistor coupled to a word line will switch when that word line switches. Bit line switching is then detected by sense amplifiers on the bit lines. Each word line is powered by a line driver which must tolerate the variations in word line loading caused by the variation in the number of bit lines coupled to the word line. For convenience word line drivers and bit line sense amplifiers are standard devices used elsewhere in ILLIAC IV. The line drivers are level converters that convert standard ECL levels of ± 0.4 volts to CTL compatible levels of ± 3.0 and 0.0 volts. The sense amplifiers are standard, ECL, 9 input, negative NAND gates.

For coupling word lines to bit lines, transistors offer several performance advantages compared to diodes or resistors. Multiple leakage paths thru resistively coupled cross points would be prohibitive in a memory

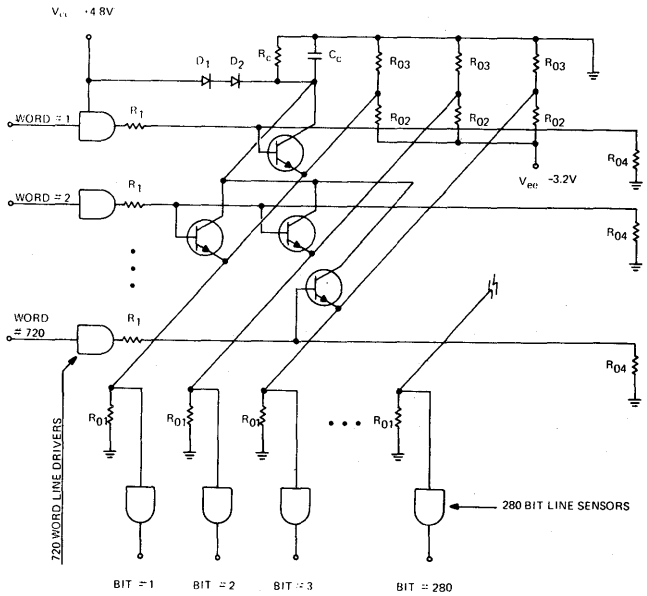


Figure 8

of this size and speed. Compared to diodes, transistors isolate bit line capacity and dc loading from the word lines which alleviates word line driving problems.

Bit lines

The required nominal input levels at the sense amplifiers are ±0.4 volts. In order to conserve power, the coupling transistors are biased to be cut-off for a low level word line. Because a bit line may be driven by a coupling transistor at any location along the bit line, each bit line is terminated at both ends in its 50 ohm characteristic impedance. The bit line terminating resistors R02 and R03 have a Thevinen's equivalent of 50 ohms returned to −0.8 volts and quiescently bias the bit line at −0.4 volts. At a high bit line level of +0.4 volts, the coupling transistor must supply 8 ma to R01 and 16 ma to R02 and R03, for a total of 24 ma. Since the V_{be} drop is about 0.8 volts, the word line is required to swing between 0 to +1.2 volts. The collector supply voltage is fed thru diodes D1 and D2, from the +4.8 volt supply, in order to reduce power dissipation in the coupling transistors.

Word lines

As explained before, because of the convenience of using available devices, the word line driver provides an output level that swings from 0. v to 3.0 v. As shown in Figure 8, R1 is inserted in the word line to attenuate the 3 volt signal to the desired 1.2 v level. Because of the large number of bit lines crossing the word lines, the length of the word line is electrically long, and is terminated at the far end in 50 ohms. The word lines are 50 ohm microstrip lines. To minimize the word line time delay each word line is divided into two segments, with each segment coupled to a maximum of 140 bit lines. Each segment is designed as shown in Figure 8 and has its own set of line drivers, and line terminations. Although there are 140 bit lines crossing each word line segment, system design requires only a maximum of 25 bit lines be coupled to any one word line segment. Each coupling transistor introduces a capacitive loading of about 1.5 pf, delays the signal propagation by about 0.15 nanosecond, and produces a peak negative reflection of about 30 mv. To prevent an excessively large reflection, no more than 3 transistors are located in succession along any word line or any bit line.

Memory partitioning

Because of speed considerations and space limitations, each word line is divided into two segments.

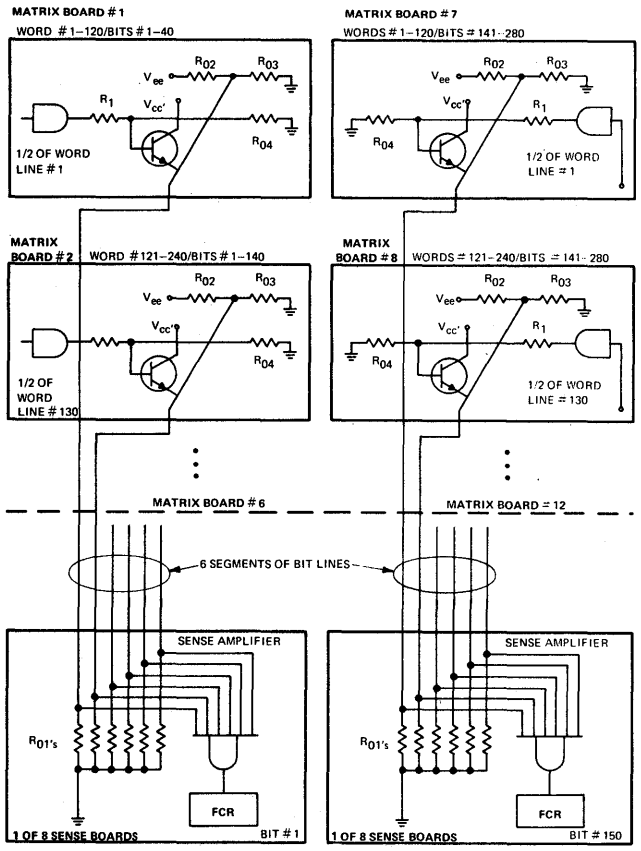


Figure 9

Similarly, the bit lines are divided into 6 segments. The result is to divide the ROM into 12 matrix boards. Each board contains 120 word lines and 140 bit lines. The sense amplifiers and output register are mounted on 8 sensing boards. Eight boards are required because of the pin limitations. The partitioned ROM is shown in Figure 9. Note that only the sense amplifiers and bit line terminating resistors (R01) are on the sensing boards. The remaining components are mounted on the 12 matrix boards. The matrix board is a special design. The sensing boards are standard 12 layer boards used throughout the ILLIAC IV Control Unit.

Mechanical description of the matrix board

The bit lines are 50 ohm strip lines which gives the advantages of low parallel line cross-talk and close impedance control. The word lines are 50 ohm micro-strip lines. Address selection lines, which are the input lines to the word line drivers, are nominally 100 ohm micro-strip lines to simplify board construction. The

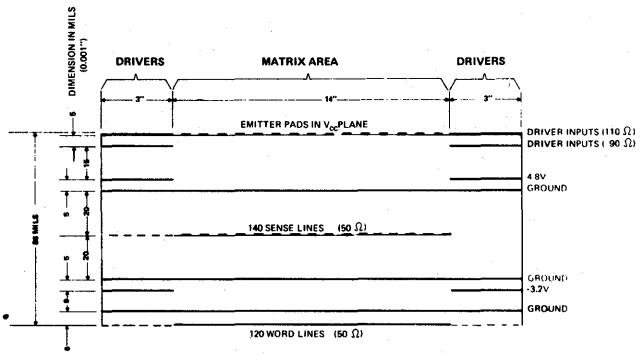


Figure 10

address selection lines are constructed in two layers with both layers using the same ground plane as the signal return path. Consequently, the two layers of address lines are slightly different in line impedance. The lines in the top layer are about 110 ohms and the lines in the bottom layer are about 90 ohms. The matrix board cross section is shown in Figure 10. The board dimensions are otherwise the same as a standard CU board, i.e., 18 inches × 20 inches.

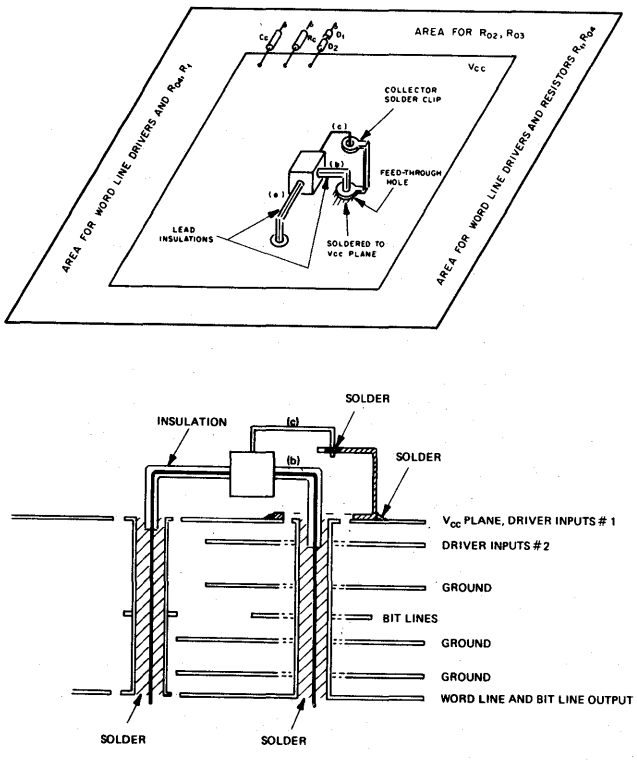


Figure 11

The transistors are cubes of 80 mils on an edge. The transistors are spaced 100 mils apart and so are the spacing of the holes and mounting pads for the emitter, collector, and base leads. The tight spacing of the transistor requires a special mounting technique as shown in Figure 11.

CHECKOUT OF MATRIX BOARDS

The coupling transistors are located on the matrix boards in accordance with the system instruction table, i.e., the output bit pattern for every word selected. The simplest checkout fixture is to apply a +0.4 volt input level to each word line driver and read the output of every bit line with a voltmeter. Such a manual checkout system would require 320 man hours for the 12 matrix boards. A more automatic checkout system is shown in the flow chart of Figure 12. The design table of the ROM is transferred onto 80-column cards. Each card will have 80 bits and 12 words. Therefore, 2 of the 80-column cards are required to complete the contents of 12 words by 140 bits, and 20 cards are required for one of the 12 matrix boards. A card reader reads one card at a time and compares the reading with the bit outputs. Any mismatches are detected and displayed by indicating lamps. With this system the matrix boards can be checked out at the rate of 9 boards per hour or 2 man-hours for 12 boards. The cost of the check out system is about $3,000.

In ILLIAC IV, neither of the above approaches is used. The Test and Maintenance Unit (TMU) area of the Control Unit has a 64-bit comparator which can be used to check out not only the matrix board but the entire area of instruction decoding. Via the I/Ø interface,

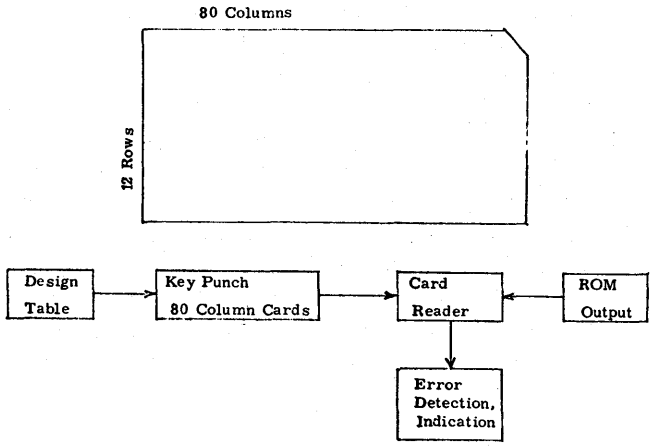


Figure 12

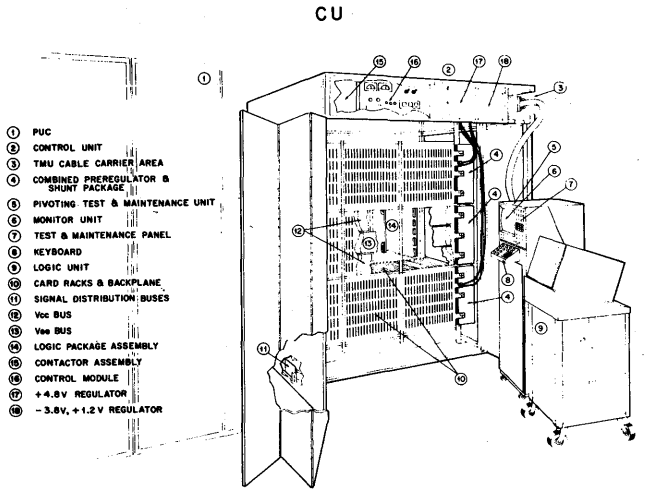


Figure 13

instruction decoding and ROM addressing can be controlled programmatically. The program generates the FINST instruction to be decoded and loads a TMU register with the expected ROM response. The comparator compares the expected and actual responses and the program proceeds if they agree. Because only 64-bits can be compared at a time, five compares are required to check each full 280 bit output of the ROM. If there is an error, the inputs being compared can be displayed on a CRT display in the TMU (Item 6, Figure 13).

APPLICATION OF MSI

For the entire function of decoding instructions into microsequences, storing enable patterns in the ROM and sensing and storing the ROM output, the present system uses a total of 28, 18 inch by 20 inch, multilayer boards. As the following example shows, an MSI approach would result in about a 50%. reduction in volume. An all MSI ROM is composed of MSI cells where each cell is M words by N bits. To reduce interconnections and conserve pinouts, x–y addressing is used to address the M words in the cell. To eliminate buffers between cells, the N bits of one cell are "wire OR'ed" to the N bits of another cell.

Control Enable ORing is no longer possible because x–y addressing generates unwanted addresses if more than one x or more than one y input is active. The number of required ROM words is increased to about 1200 because this ORing cannot be used. The amount of logic required to generate microsequences also increases. However, the generation of microsequences is now more straightforward and better adapted to MSI. Based on the average number of words required by a microsequence, the optimum number of words in a cell is determined. Assume the optimum number is 16 words. Therefore, four address lines plus one address enable line are required. The address enable permits more than one cell, i.e., more than 16 addresses, to be used in a microsequence. Because many microsequences do not use whole multiples of 16 addresses, more than 1200 addresses are required. Assume the ROM size is increased by 20% to 1440 words to account for this affect. Assume each cell has 64 outputs. This requires less than 100 pins per cell which is consistent with present MSI packaging. To obtain 280 outputs, five cells are required for every word. Therefore a total of 450 cells are required to make up the complete ROM. Allowing four square inches of surface area per cell and sufficient space for terminating resistors, bypass capacitors and connectors, a total of 8, 18 inch by 20 inch, boards are required to make up the ROM. If the remaining 16 boards required for microsequence generation and ROM output sensing and storing can be reduced to 6 boards, then the original 28 boards are reduced to 14 boards. It is important to note that in order to achieve an ROM cycle time of 50 nsec (i.e., register to register), the cell access time (i.e., address input to bit output) will have to be, approximately, 20 nsec.

ACKNOWLEDGMENT

The final ROM is the result of the efforts of many people and the authors gratefully acknowledge their contributions. Especially noteworthy is work by E. S. Sternberg in the area of logic design and T. E. Gilligan in the area of circuit design.

REFERENCES

1 G H BARNES R M BROWN M KATO D J KUCK D L SLOTNICK R A STOKES
The ILLIAC IV computer
IEEE Trans Computers Vol C-17 pp 746-757 August 1958
2 R L DAVIS
The ILLIAC IV processing element
IEEE Trans Computers Vol C-18 pp 800–816 September 1969
3 R A STOKES
ILLIAC IV: Route to parallel computers
Electronic Design Vol 26 pp 64–69 December 20 1967
4 *Multilayer printed circuit board—technical manual*
The Institute of Printed Circuits

A model and implementation of a universal time delay simulator for large digital nets

by STEPHEN A. SZYGENDA, DAVID M. ROUSE and EDWARD W. THOMPSON*

University of Missouri
Rolla, Missouri

INTRODUCTION

Although simulation of logic circuits has been attempted in the past, only those which simulate completely combinational circuits have performed with any degree of success for various types of logic. This is primarily due to the number of simplifying assumptions that can be made for combinational circuits. For example, in combinational circuits, one need not consider timing of the signals, since any given input vector will always propagate to a stable state value. Also, since purely combinational circuits do not contain internal states, the user need not define or initialize these values, as must be done (and done meaningfully) for sequential circuits. A systems study of simulation and diagnosis for large digital computing systems has been performed.[1] The results of that study have led to the implementation to be described. The model, which has been adopted, permits the user to select the level of detail most appropriate to his requirements, and thus not hamper him with overly restrictive assumptions. The advantages of this model are the following:

1. It does not require the specification of feedback loops in sequential circuits.
2. The ability to reset all feedback lines to any value at any time is not required.
3. It provides for the detection of hazards and races.
4. Simulation is not restricted to particular types of circuits.
5. Besides having the capability of simulating gate level circuits, it can also simulate at a functional level. This results in a savings of time and storage, permitting the simulation of large circuits.

Actually, three models are presented, two being subsets of the other. It is felt that these models will be useful tools in analyzing logic circuits, generating tests, and providing experience in determining a desirable level of detail for simulation.

In the next section of this paper, features are defined which appear to be desirable for a general purpose system simulator. This is followed by a description of the means utilized to achieve these desired features. In particular, a detailed discussion of the models adopted for the simulation is provided. It is felt that the heart of any simulator is the basic simulation model. The effectiveness of the simulation is directly related to the ability of the model to accurately describe the physical systems being simulated. Therefore, the adopted models are of extreme importance and will be described in sufficient detail to substantiate their effectiveness.

A description of the simulator implementation follows the simulation model discussion. (This implementation is used in a total simulation and diagnosis software system.[2]) Modes of operation and simulation optimization techniques are also described. Since a table driven simulator was implemented, a discussion of its implementation will be given.

DESIRED FEATURES OF A SIMULATION MODEL

A primary goal of this simulation package is that it possesses the ability to simulate any of the common modes of logic operation. The handling of asynchronous sequential circuits presents the most difficulty, in that circuit timing must be accurately described. This rules out the unit delay assumption used by many existing simulators (in particular, all space-leveled, compiler-driven simulators). Therefore, this model allows a variable time delay for the elements being simulated.

In the model used by Seshu[3], which is frequently used for sequential circuits, the only race analysis performed is that of checking feedback line values. All

*Present Address: TELPAR, Inc., Dallas, Texas

feedback lines are assumed to be broken at some point and a race is declared if, during any pass through the circuit, more than one feedback line changes value. One feedback line at a time is declared a winner and its value is then propagated through the circuit to determine its effect on the outputs. The race becomes critical if different stable states are reached, depending upon the order in which the feedback values were changed.

This type of model has three major deficiencies:

1. The system is very sensitive to the selection of feedback lines and the point at which they are broken.
2. A few feedback loops can produce numerous races, requiring excessive time and storage for evaluation.
3. Race analysis performed in this manner does not detect static or essential hazards.

Also, many of the races which may be detected are physically impossible. One of the goals of the model being presented will be to overcome these deficiencies.

The model used by Seshu also makes the assumption that feedback lines can be reset to any desired value at any time, even in the presence of faults. It is felt that this assumption is not necessarily valid, and the assumption is not made in the model being presented.

The speed of simulation (compilation and execution) is an important measure of a good simulator. Therefore, maximum speed is also an objective. However, the requirement for maximum speed can, and should, be sacrificed for more accurate simulation results, when required. Therefore, the speed of this simulator is a function of the level of detail required of the simulation.

Another simulation goal is the use of a minimum amount of storage. The importance of this goal is twofold. First, minimization of storage requirements will enable the simulator to handle larger circuits; and second, its use will not be limited to large computing facilities. Appropriate segmentation can help reduce the storage requirements, but only with a sacrifice in speed. For these reasons, both minimization of storage requirements and software system modularity are of primary concern.

The simulation package should be as flexible and versatile as possible, but at the same time it should be easy to understand and implement. Therefore, one need only be concerned with those options which are pertinent to the task at hand. For example, the option of simulating faults is available with or without race analysis.

An additional goal is the capability of easy adaptation of new element types. Hence, one need only supply a new description for element evaluation, and the program then adds to, or updates, the existing specifications. Therefore, this method should not require a large amount of reprocessing.

The system should also have the capability of multiple fault insertion, as well as fault insertion on inputs and outputs of functional modules.

MEANS OF ACHIEVING THE DESIRED FEATURES

There are two approaches that can be taken for digital simulation. One is the approach of modeling the entire circuit as one unit and, therefore, with one macro-model. This would be the technique used for a compiled simulator, since a compiled simulator levels and transforms the circuit into a form that can be dealt with collectively. The other approach is that of modeling the entire circuit by breaking it into smaller blocks, which can be individually modeled according to their type. This approach can be accomplished with a table driven simulator. A table driven simulator deals directly with elements, in that the circuit description is explicitly specified during simulation. Therefore, it determines, during simulation, what elements are to be evaluated next and then uses one generalized routine to evaluate all elements of any one type.

The second method was chosen for this simulator since the first contains undesirable features, such as the need of pre-leveling, location and breaking of feedback loops, inability to handle various sequential circuits effectively, etc. Although the second approach is more general, and can handle a large majority of circuit types, it does have the disadvantages of being slower and requiring more storage for certain cases. Therefore, it is these latter two disadvantages that are of great concern in this simulator structure. Solutions to these problems will be discussed later.

The ability to simulate sequential circuits is inherent in a table driven simulator, in that it dynamically levels the circuit during simulation. This is done by determining, from the circuit description, what elements need to be reevaluated due to a change in the value of a signal.

The ability to simulate asynchronous circuits relies on the ability to accurately represent the time associated with evaluation of signal values. By including the propagation time of each element in its description, and then using this parameter during simulation to order the evaluation procedure, this time factor can be accurately represented. This means that one must have the ability to accept propagation delays of different lengths instead of making a unit delay assumption.

Another side feature of a table driven simulator is that of not being required to consider feedback lines

by special, cumbersome, and inaccurate methods. These methods include explicit specification of what lines are feedback lines and the ability to reset these to any value. These two conditions are produced when an attempt is made to represent an entire sequential circuit by one model, as is done in a compiled simulator structure. The feedback specification problem is not present in a table driven simulator, in that no special case need be made concerning feedback lines. The latter case, which is commonly referred to as a reset assumption, is avoided since simulation occurs directly from a given accessible state without requiring reinitialization of the state during simulation.

As mentioned earlier, speed and storage will be a primary concern in this simulator. Three techniques will be employed to reduce these problems to an acceptable level. They are: (1) selective trace simulation, (2) parallel simulation, and (3) functional simulation.

Selective trace is a technique used in conjunction with table driven simulators which provides the ability to evaluate only those elements which have a potential of changing. For example, one need not reevaluate a gate output if all the input signals are the same as they were when it was last evaluated. Thus, simulation becomes a process of tracing changes, and their effects, through the circuit.

Signal values can be stored in one of three manners. First, one bit of a machine word could be used to represent the value of a signal. Second, each bit of a machine word could be used to represent a different signal value. Third, each bit of a machine word could represent different values of the same signal for different input vectors or different fault conditions.

The first of these three techniques is extremely inefficient in storage handling. The second approach is hard to execute in Fortran (the implementation language for the system) since it would require bit manipulation. Therefore, the third approach was taken. For this technique, n different input vectors (where n is the number of bits in the machine word length), or fault conditions, can be simulated with the same speed and storage required for the first approach. This is referred to as parallel simulation, since n unique simulations occur in parallel. The effect of this approach is to divide the required simulation time by a factor of n.

Another important implementation feature is called functional simulation. This is the grouping of a number of logic elements together and then expressing the group by its function. Thus, one need only store and evaluate the function in order to simulate the represented logic. An example of functional simulation would be the representation of an adder by storing and executing a simple add instruction, instead of storing and executing the large number of logic elements used to form an actual adder circuit. Therefore, it can be seen that functional simulation enhances simulation speed and reduces storage. The ability to implement functional simulation is compatible with a table driven simulator structure, since it models the circuit by modeling elements, regardless of the evaluation procedure used to model the element. For this reason, changing or adding element types is a simple task which involves changing only the evaluation procedure and its respective pointer.

Fault insertion is also simplified since it now becomes a matter of simply providing elements having the same characteristic as a faulty element.

When faults are automatically inserted, fault collapsing is used to reduce the number of possible faults. This is done by inserting only one fault of a group of faults which always produce the same outputs for any input combination. For example, all stuck-at-0's on the input of an "and" gate appear the same as a stuck-at-0 on the output of that gate. Therefore, the stuck-at-0's on the input need not be simulated if a stuck-at-0 on the output is simulated, since they all produce the same response and are therefore repetitious.

To further increase the accuracy of simulation, an ambiguity interval can be associated with each signal. This is a result of an inability to specify exactly when a given signal will actually make a transition from one state to the next. The requirement of an ambiguity interval comes from the fact that gates of the same type could have different propagation times. Therefore, the time delay of a gate would be represented as a minimum value plus an ambiguity region. By considering this ambiguity, race and hazard analysis can be performed during simulation.

It is not always desirable to perform race analysis, since it requires greater simulation time and storage. Therefore, there are different modes of simulation, including straight simulation and simulation with race analysis. Straight simulation will be referred to as the Mode 1 simulator, and simulation with race and hazard analysis will be referred to as the Mode 3 simulator. In order to obtain modularity in simulation and consistency in circuit modeling, Mode 1 will be implemented as a subset of Mode 3. A Mode 2 simulation is also available. This is a three valued simulation which can be used for simulation initialization. It indicates and propagates information as to whether or not a signal is defined at a given time. This is accomplished with the use of a third value. An example of the Mode 2 simulation is given in Figure 1. Here, each signal can either be 1, 0, or I (I indicates Indeterminate). Before time 0, all signals are unknown and, therefore, in an indeterminate (I) state. At time 0, A and B are changed to a 1. As a result of this change in A and B, the value of C and D must be reevaluated. C is evaluated to be 1,

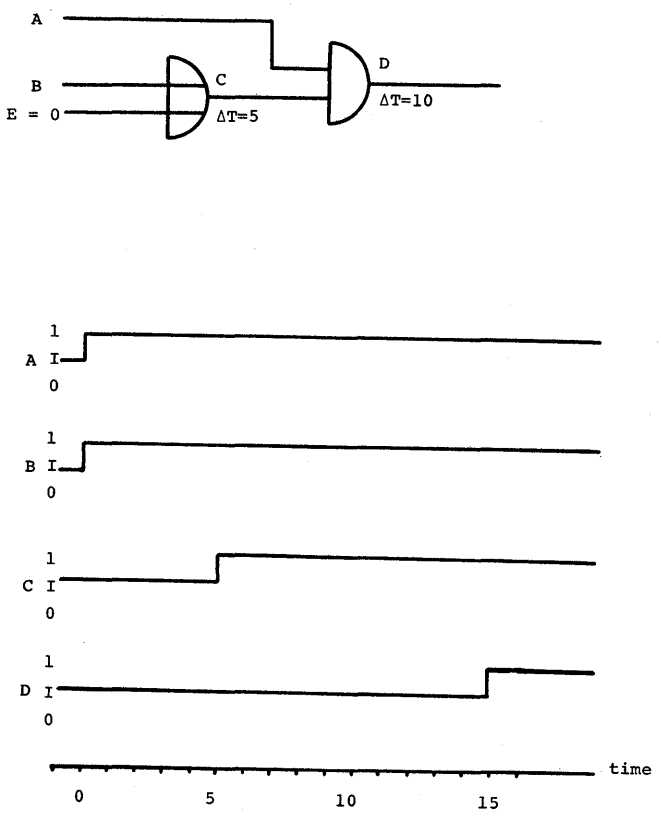


Figure 1—Mode 2 simulation

at $t = 5$. The change in A, at $t = 0$, causes a reevaluation of D. However, since $C = I$ at $t = 0$, then $D = I$ at $t = 10$ due to the change in C having not yet propagated to D. An evaluation table for C and D is indicated in Figure 2. However, the change in C at $t = 5$ causes D to be evaluated again. Hence, D becomes 1 at $t = 15$ since both C and A are known.

An example of the Mode 3 simulation is depicted in Figure 3. A and B are input signals of initial value 0 and 1, respectively. C and D are the output of inverters, which have a propagation delay of 4 and an ambiguity of 2. E, which is the set signal to an S–R Flip Flop, is the logical "and" of C and D. At $t = 0$, A changes to 1 which produces a change in C to 0 through an ambiguity region from $t = 4$ to $t = 6$. This means that the change of C could occur sometime between $t = 4$ and $t = 6$. If B changes at $t = 1$ (possibly due to an ambiguity in the circuit feeding B), then D changes value as indicated. Notice that the value of E, as a result of C and D being 1 between $t = 5$ and $t = 6$, is a potential error region. This, along with ambiguity and minimum delay of the "and" gate, produces the results indicated for E. Since E is setting the Flip Flop, this potential error region sets the Flip Flop to a potential error value, which is the resulting state of Q. From this example, it can be seen how the ambiguity in propagation delay is handled, as well as how Mode 3 simulation handles potential error regions and how these potential error regions can result in essential hazards.

In general, the mode 3 simulator is used to propagate potential error regions to provide determination of the existence of essential hazards. The unique characteristic of the Mode 3 simulator, that does not exist in the other modes, is that it carries regions instead of

and	0	1	I
0	0	0	0
1	0	1	I
I	0	I	I

a) Logical "and"

or	0	1	I
0	0	1	I
1	1	1	1
I	I	1	I

b) Logical "or"

Figure 2—Logic table for gate evaluation in a three value simulation

single values. A technique similar to this has been described by D. L. Smith.[4] A spread is depicted with the use of a New Value (NV), as well as a Current Value (CV), along with the Potential Error value (PE). During simulation, this ambiguity region is represented by simulating the earliest possible transition point (the CV) and the latest possible transition point (the NV). The potential error is then a logical function of the CV, NV, and the PE. With respect to the simulator, this evaluation procedure simply appears as another element type.

Sequential logic circuits containing global feedback loops are extremely difficult to simulate, even for the simplest of design philosophies. Whether accurate simulation is achieved, most often depends upon the design employing a special type of sequential action, which is consistent with the particular simulator being used, or the person simulating the circuit must have a very intimate understanding of the circuit operation. These two circumstances are more often the exception rather than the rule. In many design environments, as a result of practicality, as well as necessity, these conditions are rigidly forced upon the user by their simulation structures. In order to alleviate this problem, the largest possible user flexibility was a goal for this simulator. One of these degrees of freedom is presented in the following example, which shows how race analysis of an asynchronous sequential circuit can be performed.

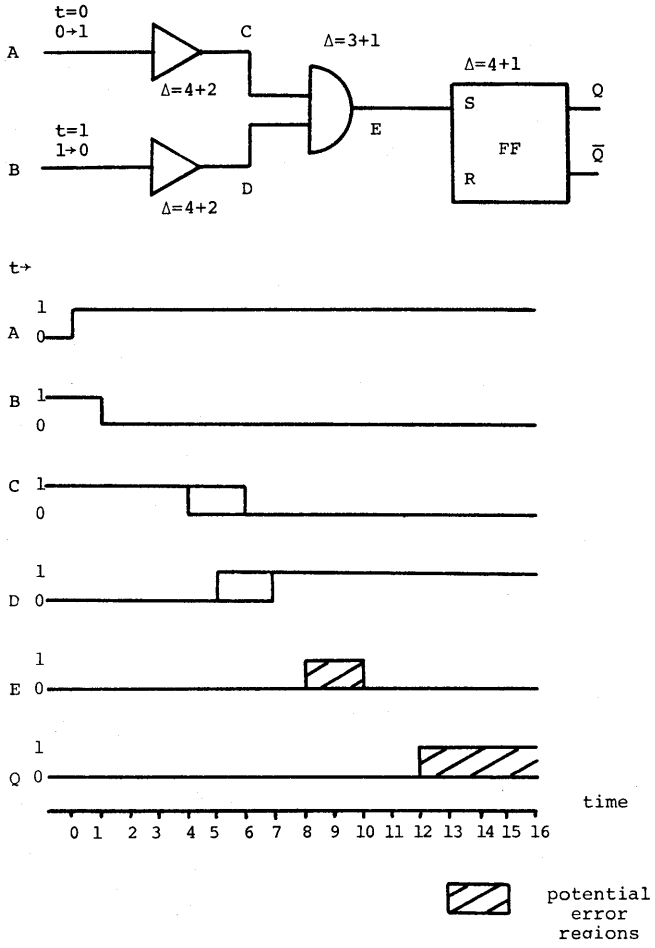


Figure 3—Mode 3 simulation

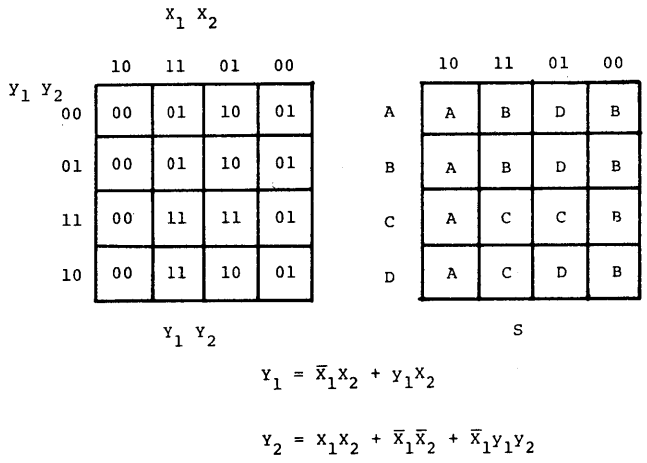


Figure 4—A transition table and state table for a simple sequential circuit

A Transition Table and State Table are given in Figure 4 for a simple sequential circuit which has been implemented in Figure 5. A race can be seen to exist between states C and D for the input vector 01 and stable state B. Whether this race is a result of improper design, characteristics of a circuit containing a faulty element, or an intentional risk, is unimportant. The important thing is that the simulation of this circuit is capable of revealing sufficient information to determine how the circuit will, or could, act when physically implemented. Figure 6 shows the response of X_1 changing from a 1 to a 0 according to the unit delay assumption (the circuit is in stable state 01, with $X_1X_2 = 11$). From Figure 6, it can be observed that the circuit makes a transition from state B to state D, a seemingly definite and satisfactory result.

By considering the circuit response as indicated in Figure 7, which uses more accurate delay time information (as given by the minimum delay in Figure 5) for each gate, it is observed, through this type of simulation, that all isn't as simple as indicated by the previous simulation. It can be seen that, as the accuracy of propagation time becomes closer to that of the physical circuit, the race between states D and C comes closer to actuality. It appears here as the simultaneous transition of Y_1 and Y_2. This type of simulation is one which might be performed by Mode 1 simulation.

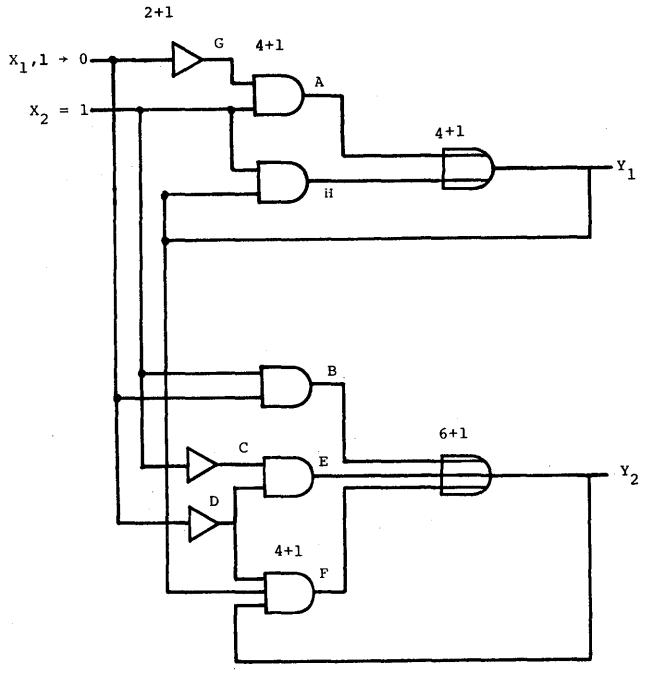


Figure 5—Example sequential circuit

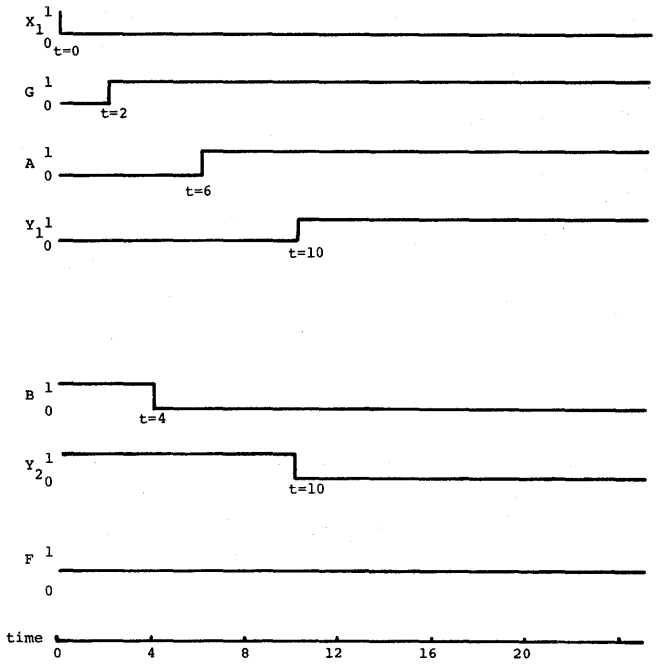


Figure 7—Results for circuits with variable time delays

The curiosity raised by using a more accurate representation for the delay time can be satisfied by also considering an ambiguity time (as indicated in Figures 5 and 8) associated with each gate. The state variable Y_1 could change anywhere between $t = 10$ and $t = 13$. This is a result of the possible variation in time delays

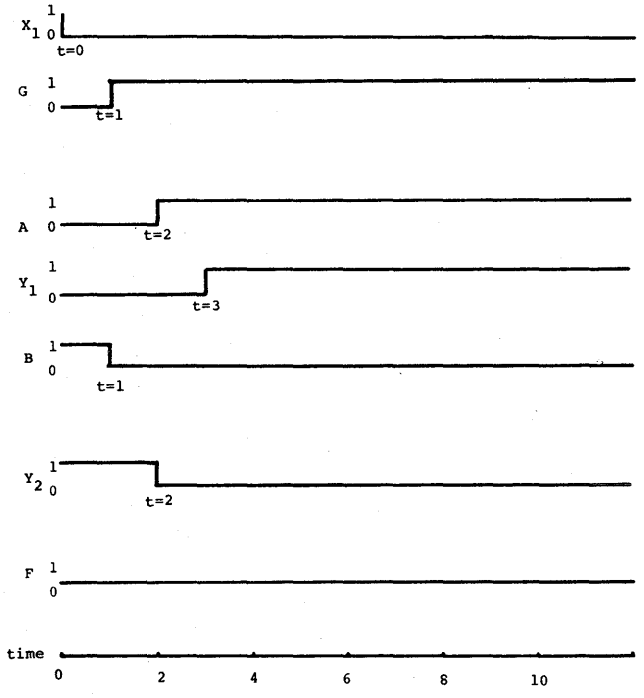


Figure 6—Results for circuits with unit time delays

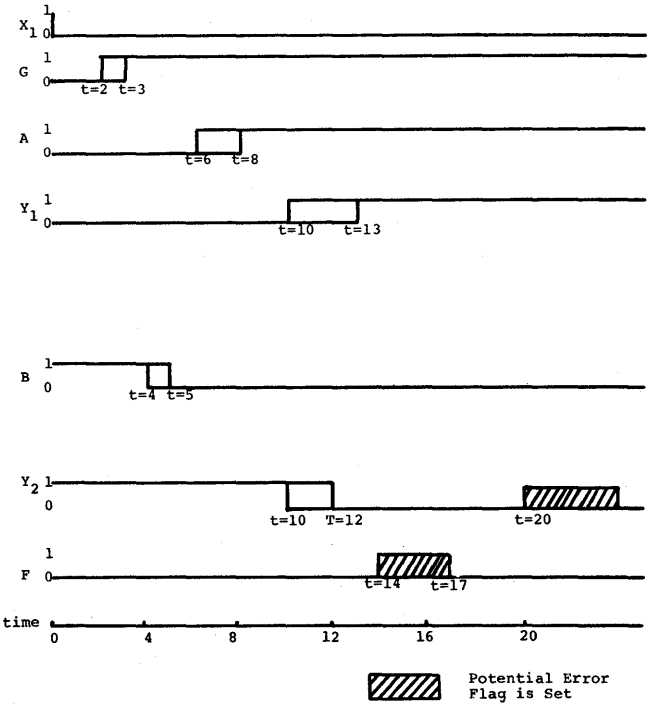


Figure 8—Results for circuits with variable time delays and ambiguity

of the inverter, "and" gate, and "or" gate, along the propagation path X_1. Similarly, Y_2 could change between $t = 10$ and $t = 12$. The value of F is essentially the "and" of Y_1 and Y_2, since $\bar{X}_1$ appears as a constant 1. However, the "and" of the two ambiguity regions for Y_1 and Y_2 is not only another ambiguity region in F, it is also a potential error region as well. In actuality, F may or may not produce the momentary 1 spike between $t = 14$ and $t = 17$. Note that a transition region is concerned with the question of *when* a transition will take place. However, a potential error region is concerned with *whether* a transition could take place.

Thus, from this example, it can be seen that a variation in propagation delay is enough to produce a critical race condition from a seemingly stable design. This condition is detected in Mode 3 simulation when the potential error flag is set for the state variable Y_2, as indicated by the shaded area in Figure 8.

This example demonstrates some of the problems that could be encountered when simulating sequential circuits. It also shows how these problems can be handled through the various modes of simulation available in this simulator.

To use these three modes of simulation, one need only specify which mode is desired, so that the appropriate evaluation routines will be used. Since the only difference is in the evaluation routines, the same circuit description can be used for Mode 1, 2, or 3 simulation.

One important feature of this approach, with respect to race analysis, is that race analysis occurs concurrently for nested races. Therefore, only one simulation must be performed for n nested races, as compared to as many as 2^n simulations for other approaches.

SYSTEM IMPLEMENTATION

The first major implementation decision was the choice of a programming language in which the simulator would be written. Assembler, Fortran and PL/1 were considered. Utilizing an assembler language could result in a little faster execution, with somewhat less storage required. However, Fortran was chosen since it would be easier to implement and is considerably more machine independent. Although PL/1 has some seemingly desirable features, they were sacrificed for the more commonly acceptable Fortran and the small decrease in execution time and storage. It was also desirable for this simulator to be acceptable for use on smaller machines, which have limited storage and compiler facilities. For these reasons Fortran was considered more desirable than PL/1.

The basic simulator consists of three tables, the Time Queue Table (TQ), the System Description Table (SDT), and a table which contains the Current Value of each signal (CV). The time queue table contains events that occur at time t, where t is the index of the time queue table. The system description table contains pointers to the evaluation routines used to determine the output values of the element, pointers to the fan in and fan out, and also contains the number of fan outs for each signal.

Using these three tables, simulation is performed as follows:

1. All values that exist in the time queue, at the current simulation time, are transferred to the current value table, thus causing any projected changes in value to take effect.
2. If the new value entered in the current value table is different from the old value, then all elements that are immediately affected by this change are reevaluated. (This is accomplished by following a fan out list.)
3. The results of these reevaluations are projected into the time queue at the current time plus the minimum propagation delay of the signal.
4. The current time is incremented until an entry is found in the time queue, and then the process is repeated again.

This process is restated in a flow chart form in Figure 9.

In addition to this basic structure, other tables are used for optimization of both speed and storage. For example, functions are evaluated indirectly, via the Function Description Table (FDT), which also specifies additional parameters used in the evaluation routine. These parameters are: function type, time delay, number of inputs, and bus length. This permits a minimum number of evaluation routines that must be provided for simulation. By use of the FDT table, the same routine would be used for a 2 input "and" gate with a time delay of 5, as would be used for an 8 input "and" gate with a time delay of 8.

To keep the size of the TQ from becoming prohibitively large, a Macro Time Queue Table (MTQ) was implemented to store events which occur at large time intervals, relative to the largest propagation delay for any gate. The Time Queue was then made cyclic in coordination with the MTQ, where each cycle of the TQ advances the MTQ one step.

Some gate level elements, such as flip-flops, as well as functional elements, have multiple outputs. To be able to simulate this type of element, an additional entry is provided in the SDT Table, which is used to chain the output together.

An ultimate goal of this system simulator is to

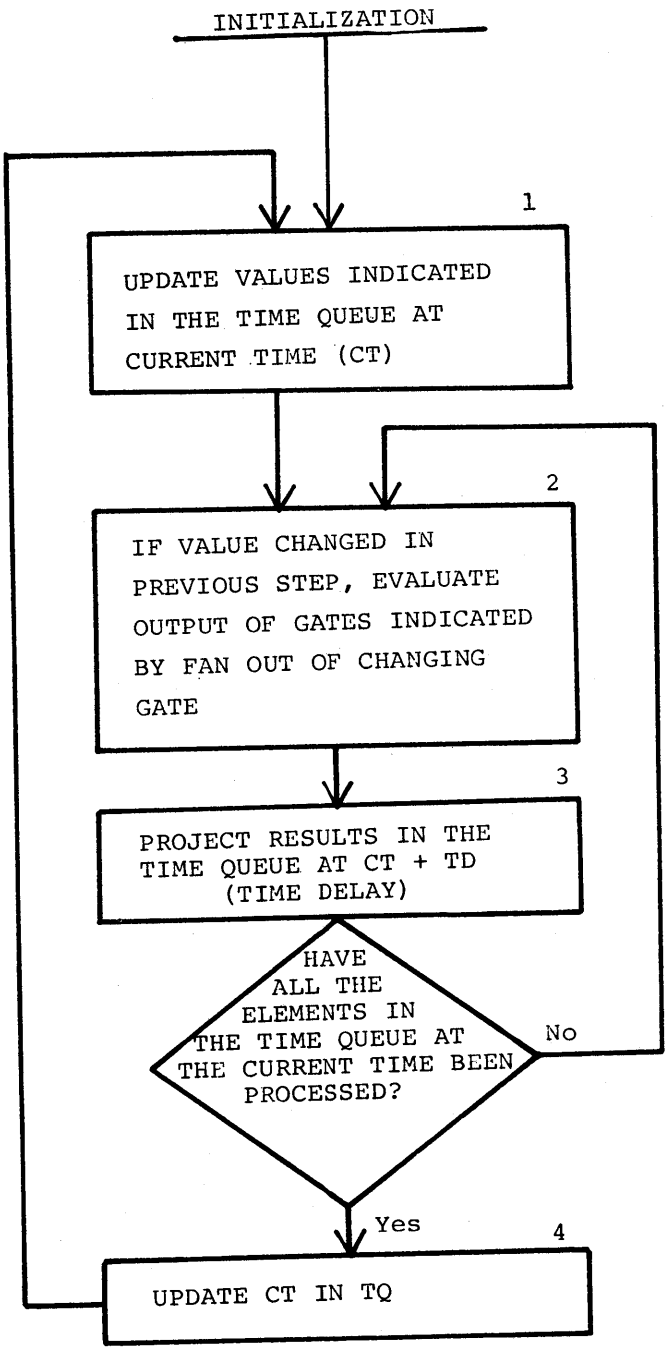


Figure 9—Flow chart of simulator

possess capability of simulating elements other than actual gates, such as functional modules.[5] Since functional modules are just as apt to be dealing with busses as with single lines, the ability to specify bus lines collectively would make functional module specification an easier, as well as a more meaningful task. For this reason, the capability of specifying busses collectively has been implemented with the use of a paging scheme, where Bus Value (BV) is a group of pages and Bus Value State (BVS) is a table which indicates use, length and location. Therefore, the CV of a bus is an indirect pointer to a page, which contains the actual values of the bus signals.

It can be seen that the System Simulator is independent of the type of function used to evaluate the output signals of the element. For this reason, any type of element can be simulated which can be described in a discrete value system. Therefore, the power of module simulation is directly proportional to the kinds of module descriptions which are permitted.

In an attempt to cover the widest range of module descriptions, five types of descriptions are permitted. These are: (1) gate elements, (2) standard functional modules, (3) compiled gate modules, (4) computer design language modules, and (5) Fortran modules.

Evaluation procedures for gate elements are defined by gate type. Thus, one can specify gate elements, to the system simulator, by giving the gates fan in and fan out. This would be used primarily for circuits which could be specified at the gate level.

Similar to the gate modules are the standard functional modules, in that they are predefined routines which can be used by specifying an element as a standard functional module type. An example would be an n-bit 2's complement adder. Thus, to define a complete adder, one need only give its function type, the number of bits being added, and its time delay. This feature was provided in order to eliminate much of the trivial task of redefining common functional modules, and also to provide faster, more efficient, system routines.

To permit the initial design specification to become the initial functional representation for design verification on a macro-level, a computer design language is allowed for module description. This is done by compiling this description into an equivalent Fortran subroutine, which has the inputs equivalenced to the appropriate current value table, and the outputs queued in a scratch array. In order to provide sequential control modules at a design language level, sequential modules can be described in a flow table form of expression. Thus, sequential control variables can be generated by such a sequential module.

Also, to provide the capability of compiled simulation, one can specify modules at the gate level and have these transformed into compiled code for simulation. This is desirable for purely combinational modules which can be simulated faster, or with less storage, in a compiled simulation fashion.

Fortran module descriptions can be used as a means of generating new, efficient, standard functional modules. They can also be used to generate special modules

whose function cannot be described easily by a high level system design language.

To make this system more usable, one must have a means of redefining modules without requiring the reprocessing of the complete system description. This is done by automatically defining boundary elements for each module of the system.

A boundary element is an element which is placed at each input and output of a module to isolate that module from the rest of the circuit. Each boundary element has one input and one output, and the output value equals the input value. Taking this approach permits changes in the module definition without changing descriptions outside the module. This is depicted in Figure 10 where B1, B2, B3, and B4 are boundary elements for module M2. Gate G1 fans out to G2 and B1 when module M2 is expressed functionally. But, if module M2 is expressed at the gate level, without boundary elements, then the fan out of G1 would be to G2, G3, and G4. Without boundary elements, it would be necessary to change the fan out description of G1 (which is outside the module), if the user wants to change M2 to a gate representation. But, with the boundary elements inserted, the fan out of G1 (to G2 and B1) is still the same independent of the type of expression used for M2. This could be of extreme importance, when a number of design groups are using the same high level description for the complete system, while considering their own particular section at the gate level.

For fault insertion, two tables are used. The Fault Table (FT) indicates the type of fault and which leads of the gate with which it is concerned. The Logical Fault Mask Table (LFMT) indicates in which bit (or subject machine) the fault is present.

During Mode 1 simulation, the value of each signal is stored in the full word array CV. The values in the CV are updated by transferring the appropriate value, which is indicated indirectly in the time queue, to the CV, at the time indicated by the time queue description.

The same simulation structure exists for Mode 2 simulation, with the use of different evaluation routines for element output evaluation. An Indeterminant Value Array (IV) is used for storing the third value necessary for Mode 2 simulation.

Due to the necessity of being able to process time intervals, in Mode 3 simulation, a few minor modifications must be made in the simulator structure. This, however, is not apparent to the simulator user. Storage must also be provided for the Current Value (CV), New Value (NV), and the Potential Error (PE), along with the more detailed evaluation procedures used to determine these quantities for each element.

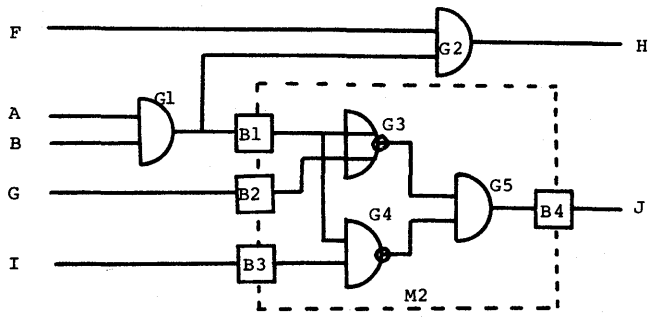


Figure 10—Module boundary elements

CONCLUDING REMARKS

The simulator described in this paper has been programmed in Fortran IV on an IBM 360/50, with 256K bytes of core storage, at the University of Missouri—Rolla. The size of a system that can be simulated is a function of the available memory capacity of the host machine. For 256K bytes of storage, approximately 3000 elements could be simulated, using the Mode 1 phase of the system. (An element can range from simple gate elements to complex functional elements.) This would be reduced to approximately 2000 elements for Mode 3 simulation. For trial simulation runs, a running time of 100 μs/pass·fault·element has been obtained, utilizing Mode 1 simulation. This, however, is fairly circuit dependent.

It is felt that this simulator represents a uniform systems approach to simulation and diagnosis. Versatility of the models utilized has resulted in this capability. The system not only allows various types of simulation for different hardware implementations, but also provides the ability to handle the total system as a collection of subsystems. Thus, each subsystem can be simulated according to the particular type of circuit involved, the requirements imposed upon the circuit, and the most applicable simulation technique for the particular subsystem.

REFERENCES

1 S A SZYGENDA
A software diagnostic system for test generation and simulation of large digital systems

Proceedings of the National Electronics Annual Conference December 1969

2 S A SZYGENDA
TEGAS—a diagnostic test generation and simulation system for digital computers
Proceedings of the Third Hawaii International Conference on System Sciences January 1970

3 S SESHU
The logic organizer and diagnosis programs
Report R-226 Coordinated Science Laboratory University of Illinois Urbana Illinois (AD605927) 1964

4 D L SMITH
Models and data structures for digital logic simulation
Masters Thesis Massachusetts Institute of Technology 1966

5 D M ROUSE S A SZYGENDA
Models for functional simulation of large digital systems
Digest Record of the Joint Conference on Mathematical and Computer Aided Design October 1969

UTS-I: A macro system for traffic network simulation

by HOWARD LEE MORGAN

Cornell University
Ithaca, New York

INTRODUCTION

One of the major crises which our cities must face is the problem of traffic congestion. Already, many areas are so congested that new roads seem to be the only answer. In densely populated areas, however, new roads are often unwanted because of the valuable land they use and are not the solution to today's problems because of the long time delay between planning and construction. Therefore, it is essential that cities obtain maximum throughput from existing roads before new construction is tried. A general urban traffic network simulation system has been designed and programmed to assist traffic engineers and planners in studying alternative solutions to traffic problems. The implementation consists of a set of macros which are used to describe the network, and a set of subroutines which perform the actual simulation.

Many analytical approaches to traffic control problems have been tried with varying degrees of success.[1] Queuing and other applied probability models have led to the progression system of light control, one of the more successful methods of smoothing traffic flows. Mathematical programming has been used to optimize some of the criteria for smooth flow, e.g., number of stops, or total delay time. Both of these techniques, however, have limited usefulness when applied to specific large networks because of complicating real world factors such as garages emptying onto streets or variations in driver behavior. These and other analytical methods have usually been applied to systems which had only static methods for signal light control. The technology now exists for interactive computer controlled signal lights. Sensors placed on the streets can report to a central computer which can analyze the traffic pattern and send control signals back to the lights in real-time. Mathematical analysis of these systems is extremely difficult.

The UTS–I system is a general traffic network simulator which has been developed to permit one to examine the effects both of "firefighting" techniques such as one way streets, elimination of turns at congested intersections, reversing directions of streets during peak hours, and major improvements such as adding new roads or installing computer controlled signal systems.

UTS–I is a microscopic simulation model in that the basic transaction unit is an individual vehicle. This type of model more closely approximates the real world behavior of a specific traffic network than the macroscopic model, which would consider groups of vehicles as one unit. While the macroscopic model may be more efficient with respect to run length when general systems are being studied, the microscopic model should yield more insight into particular trouble spots in a specific real system.

Any combination of intersections and distances between intersections can be specified. Such features as three-way and larger intersections, stop or yield signs, timed or sensor controlled traffic signals, or any combination thereof can be specified at any particular intersection. Traffic light sequences can be set for individual intersections, or light controls can be interconnected with other intersections and traffic flows. Other traffic simulation programs[2,3,4] have included some of the above features, but rarely all in one system. In addition, describing the network to be simulated has been as much a problem as writing the simulator itself.

The basic flow of a car through the system is simple. A car appears from a source and advances to an interaction point (sensor, intersection, end of queue, or slower moving vehicle). Eventually, the car steps up to the intersection and a random number is generated to determine if a turn will be made. When the car has the right of way and the intersection is not blocked by another vehicle, it advances through the intersection and onto the next road segment. It is then assigned a

new velocity which it maintains until it reaches the next interaction point. In this way a vehicle is stepped through the system until it departs on one of the exit roads.

The inputs to the system are arrival distributions at all points where cars may enter the system and turning percentages at each point where a directional option is available to the vehicle. Information related to driver behavior, e.g., wave delay time, velocity, etc., must also be input. A portion of this information may be common to all intersection models while some of the information may depend on specific details of a particular system.

IMPLEMENTATION

UTS–I is actually a GPSS/360 program which makes heavy use of the MACRO facility. GPSS was chosen for the initial UTS implementation because of the availability of the processor and the fact that experienced GPSS programmers were available. Because of the modular design, the simulator can be easily reprogrammed in any digital simulation language, and is now being rewritten in SIMSCRIPT–II, which offers more flexibility in the use of macros, in input and output formatting and, in the basic simulation scheduling philosophy than does GPSS.

The system consists of several subroutines, each of which simulates a particular subsystem of the network, two important matrices which drive these subroutines, and a description of the flow through the system provided through the use of macros.

The main routines are:

1. Road Segment System—simulated by LINK
2. Intersection System—simulated by ISECT
3. Interaction Point System—simulated by IACTP
4. Intermediate Segment System—simulated by INTER

The vehicle enters the *road segment.* This routine checks to see if the vehicle has caused a segment overflow. If it has, the previous intersection is blocked and will remain blocked until the segment overflow condition is alleviated. The vehicle then proceeds to the end of the segment's queue at its own preferred velocity or at the velocity of a slower vehicle preceding it (after catching up to the vehicle preceding it, and remaining at a reasonable distance behind).

Upon entering the *intersection system* the vehicle enters the road segment's queue and waits. As the vehicles preceding it move through the intersection, this vehicle moves up in the queue. Eventually, it is first on line, i.e., it is in "control" of the segment's queue. At this time the vehicle checks to see if it has the right of way. If it does not, it waits until it does. When it does have the right of way, the vehicle proceeds across the intersection. This causes a wave of car movements to be propagated through the queue. If the segment overflow condition exists, the previous intersection will eventually be unblocked. The vehicle after moving through the intersection, turns onto the next road segment.

The vehicle may come to an interaction point, which is a point on its lane where the vehicle interacts with vehicles on other lanes. The vehicle will not reach the interaction point until its position in the queue reaches that point along the segment. As vehicles leave the road segment, this vehicle moves up until it crosses the interaction point. At this time, the proper interaction is caused. Interaction points may be used to simulate flow at a point where one lane splits into two lanes.

The *intermediate segment system* is used in the same manner as the Road Segment System, except that it is used only between two interaction points or an interaction point and the end of the segment's queue.

The input data for UTS is organized into two matrices. These are the intersection entry matrix and the distance matrix. A vehicle crossing an intersection will have to cross a number of areas at which conflicts with other traffic flows could occur. Each of these areas is called an Intersection Conflict Cell (ICC) and is assigned an identification number. ICC's for each possible path across the intersection are stored in successsive columns of a given row of the intersection entry matrix. Vehicles waiting at "STOP" or "YIELD" signs interact with vehicles on the road to which they must yield. There is an area on the road to which the vehicle must yield which may not contain a vehicle if the vehicle at the stop or yield sign is to enter the intersection. This area is called a Continuous Interaction Area (CIA). Each CIA has an identification number which is also stored in the intersection entry matrix. The same scheme used for ICC's and CIA's is used for traffic lights. The number of lights which must be green and the location of the first of these are stored in the intersection entry matrix.

The distance matrix is used by the model for storing the following data:

1. Lane length.
2. Distance to interaction points from the beginning of the lane.
3. Velocity of the vehicles which have just passed the interaction points.
4. The length of the vehicle which is at the head of the queue.
5. The total length of all vehicles which have not yet reached the queue.

```
        SEGMT       SUBROUTINE
SEGMT   ASSIGN      2,P11                   SAVE ROAD ID #
        ASSIGN      3,P12                   SAVE CAR VELOCITY
        MSAVEVALUE  2+,P2,11,P7,H           SAVE CORRECT ADDITIONAL LENGTH
        TEST LE     V$OLEN,KO,NBLK          SHOULD OTHER ROADS BE BLOCKED?
        SPLIT       1,BLOCK                 SEND OFF BLOCKING TRANSACTION
NBLK    ASSIGN      9,KO                    SET DISTANCE TRAVELLED TO 0
        ASSIGN      19,P13                  SET MATRIX2 POINTER
        ASSIGN      5,C1                    RECORD TIME
ROAD    ASSIGN      4,V$OFFST               SET MATRIX2 POINTER
        TEST L      MH2(*2,2),MH2(*2,*4),   POINT  WHAT'S NEXT IACTP
        ASSIGN      19,K1                   SET MATRIX2 POINTER
        LINK        V$CHAIN,FIFO,LANE       WAIT UNTIL ROAD IS CLEAR
LANE    TEST GE     MH2(*2,2),P9,ATQUE      HAVE WE REACHED THE QUEUE?
        ASSIGN      18,MH2(*2,2)            SAVE DISTANCE TO QUEUE
        ASSIGN      4,K2                    SET MATRIX2 POINTER
MOVE    ASSIGN      8,V$VEL                 RESET VELOCITY
        MSAVEVALUE  2,P2,6,V$VEL,H          SAVE CORRECT VELOCITY
        ADVANCE     V$DIFR                  IF VEHICLE IS EARLY, ADJUST IT
THERE   ASSIGN      5,C1                    RECORD PRESENT TIME
        ASSIGN      9,P18                   SET NEW DISTANCE TRAVELLED
        TEST L      P18,MH2(*2,2),ATQUE     HAS QUEUE LENGTH CHANGED
        ASSIGN      18,MH2(*2,2)            IF YES, SAVE NEW QUEUE DIST.
        ASSIGN      19,P13                  SET P19 FOR OFFST
        ASSIGN      4,V$OFFST               SET MATRIX 2 POINTER
        ASSIGN      19,K1                   SET P19 FOR END OF QUEUE
        TEST L      P18,MH2(*2,*4),POINT    HAVE WE REACHED THE IACTP?
        TRANSFER    ,MOVE                   IF NOT, CONTINUE DRIVING
ATQUE   MSAVEVALUE  2-,*2,2,P7,H            ADJUST QUEUE SIZE
        MSAVEVALUE  2-,P2,11,P7,H           READJUST ADDITIONAL LENGTH
        ASSIGN      19,K1                   SET CHAIN INDEX
NOSEN   UNLINK      V$CHAIN,LANE,1          CLEAR THIS PORTION OF ROAD
        TRANSFER    P,6,1                   RETURN
```

Figure 1—SEGMT subroutine

The method of implementation will be made clearer by a more detailed look at the macro and subroutine package, and by an application.

MACRO AND SUBROUTINE PACKAGE

This section deals with the macros and subroutines which make up the UTS–I system. The reader who is unfamiliar with GPSS/360 may skip this section without loss of continuity.

Each of the four UTS–I macros, LINK, ISECT, INTER, and IACTP, all generate a similar sequence of GPSS code. This code consists of saving the values of some parameters, setting some new parameters for the particular transaction, and calling an appropriate subroutine. For example, the LINK macro, which is used for vehicles which have just turned onto a road segment, might be written as follows:

LINK MACRO #A, #B, #C, #D, #E

where #A is the lane ID, #B is the preferred velocity, #C is the expected next interaction point order number (1 if a queue is expected next, 2, 3 or 4 if any other interaction point is expected), #D is the ICC ID to be used if the lane overflows, and #E is the time which it would take for the vehicle to start up and move from ICC #D, if it were backed up into ICC #D.

This usage of the macro would generate the following GPSS code:

```
SAVE MACRO #A, #B, #C, #D, #D
                                (saves parameter values)
     TRANSFER SBR,SEGMT,6
                                (transfers to subroutine)
```

The SEGMT subroutine is shown in Figure 1. The flow through this subroutine is simple. First, it tests to see whether or not this vehicle will cause the lane to overflow. If so, certain other feeder roads are blocked, by sending out transactions to block these roads. When the road is clear, the vehicle is advanced to either the end of the queue or to the next interaction point along this road segment. The queue is maintained as a user chain for efficiency. When the vehicle enters the queue, its length is added to the length of the queue, and the available distance on the road is adjusted accordingly.

Each of the other macros calls a similar subroutine to handle the actions associated with it. Listings of these subroutines and macros are available from the author.

AN APPLICATION

UTS–I has been used to study traffic flow in a three intersection, 24 lane network near the Cornell University campus. Figure 2 is a schematic diagram of this bottleneck area.

Arrival rates were determined by counting cars in continuous two minute intervals between four-thirty and five-fifteen p.m. Each road was measured in the same manner so as to keep any bias constant. The percent of the vehicles turning in each direction was determined by counting absolute numbers of cars following each distinct path at each intersection. Segment and queue distances were measured alongside the road with a fifty foot tape measure.

The variables related to driver behavior were estimated as follows:

MWAVE—the time for the vehicle startup wave to reach a specific interaction point or the previous

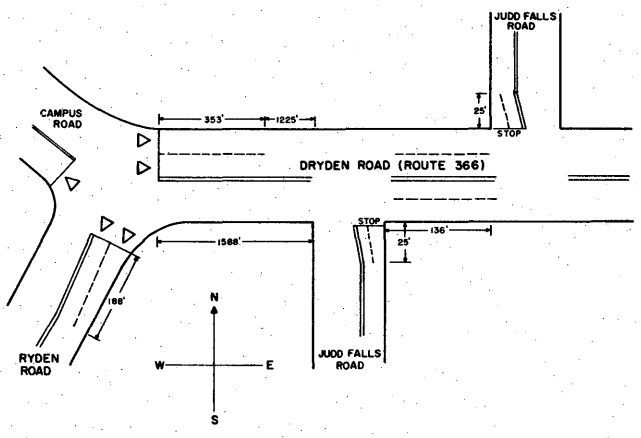


Figure 2—Diagram of the simulated system

intersection was estimated as:

$$2 \text{ seconds} + \frac{L - D_i}{150 \text{ ft./sec.}}.$$

SWAVE—the time for the vehicle stopping wave to reach a specific interaction point or the previous intersection was estimated as:

$$2 \text{ seconds} + \frac{L - D_i}{200 \text{ ft./sec.}}$$

where L = the road length and D_i = the distance from the beginning of the road to interaction point i.

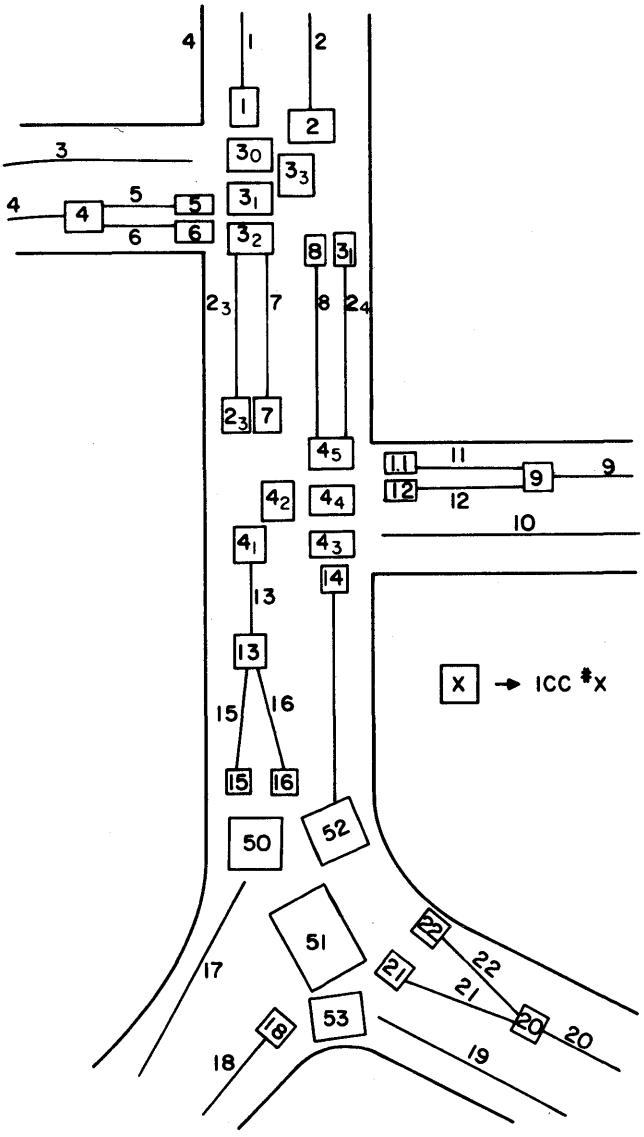


Figure 3—Intersection conflict cells and lane numbering

THE SYSTEM MODEL

```
RODI    GENERATE    8,FN$EXP,,,,20,H
        ASSIGN      1,FN$TRN1
        ASSIGN      7,FN$LNTH
LINK    MACRO       1,59,2,0,0
IACTP   MACRO       K1,2,1
INTER   MACRO       B
IACTP   MACRO       K1,3,2
INTER   MACRO       1
ISECT   MACRO       1,FN$NAME,FN$PATH,1,0,0,0,0,2
        LEAVE       1,1
        LEAVE       2,1
        TRANSFER    P,6,1
ROD2    TABULATE    1,1
        TERMINATE
ROD3    TABULATE    1,1
        TERMINATE
```

Figure 4—Simulation for road number 1

Figure 3 shows the system which was simulated with all lane numbers, ICC's, and CIA's marked. From this diagram the intersection entry matrix can be filled in. For example a vehicle in lane 1 may cross the intersection to either lane 3 or lane 7. If it goes to lane 3, it must cross ICC 30. If it goes to lane 7, it must cross ICC's 30, 31, and 32. One can make the matrix more compact if 30 is used for both types of crossing. Thus we get the following values:

$$\text{IEM}(1, 1) = 30$$

$$\text{IEM}(1, 2) = 31$$

$$\text{IEM}(1, 3) = 32$$

The flow of vehicles along the roads in the system is specified by writing a sequence of macros (with some interleaved GPSS) which generate subroutine calls to the appropriate routines. For example, the road marked #1 in Figure 3 is simulated by the code shown in Figure 4.

The following outputs have been obtained from the model:

1. Statistics for transit times for the entire system.
2. Average waiting times at STOP signs.
3. Queue statistics.
4. Lane utilization.
5. Graphs of lane utilization, histogram of transit times, and cumulative distribution of transit times.

This output is supplied at intervals of 20 simulated minutes for one hour of simulated time.

An example of the lane utilization graph is shown in

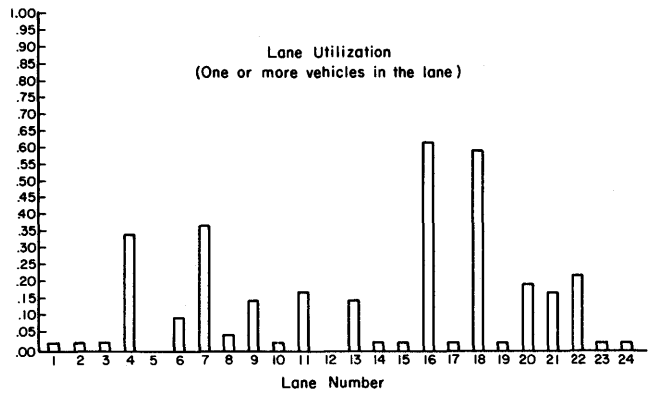


Figure 5—Lane utilization graphs

Figure 5. Lane utilization is the total time vehicles were "in control" of the lane divided by the total simulation time up to that point. This figure is thus the proportion of time that vehicles occupied the lane and the lane was blocked. This and other statistics have been compared with real world data and the model has proved a valid representation of the real system. Experiments which are now under way are attempting to discover a good traffic control algorithm for this specific network. Sensors are installed on these streets and a computer controlled network will be simulated.

USER INTERFACES

The user interface is one of the most important parts of any system. Since simulation programs are often quite sophisticated in their use of list processing and other non-FORTRAN oriented techniques, the average engineer who is not a computer specialist often has a hard time constructing or using a simulation model. While the macro subroutine system described above makes traffic network simulation easier to program, the author feels that much can still be done to improve the usability of the system both by traffic engineers, and computer programmers.

To this end, a simulation generator program is now being written. The input to this program will be a map of the traffic network to be simulated, with needed parameters marked directly on this map. The output of the program will be the two matrices and the sequence of macros which are used to describe the system. An example of the input form which will be used is shown in Figure 6. At present, this will be keypunched by laying a grid over the form, and then translated by the program into the required network.

During the seventies, when graphical input/output

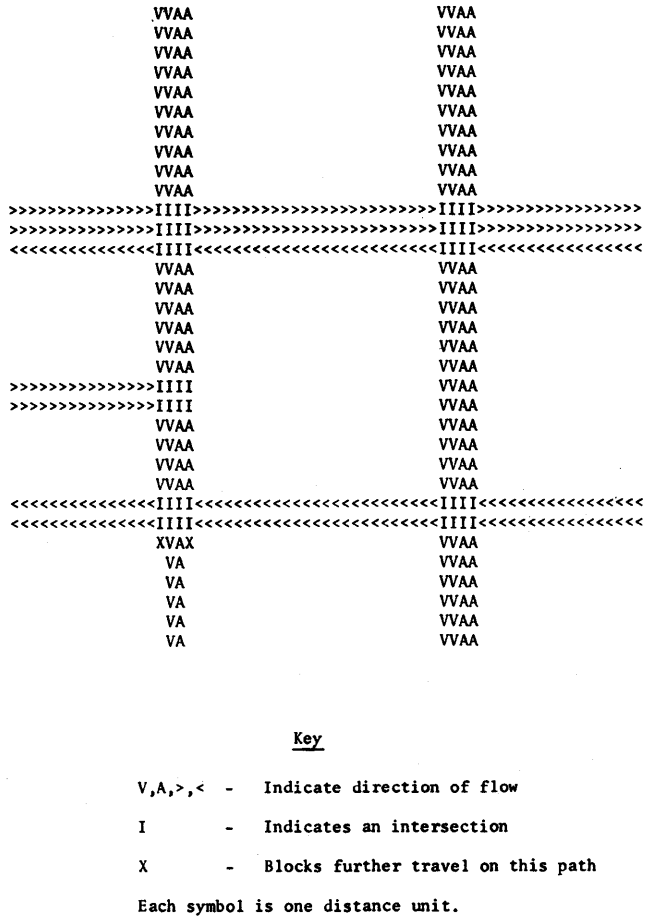


Figure 6

devices are more readily available, it is hoped that this map can actually be displayed on a CRT, and parameters altered during the simulation. Systems of this type have been discussed,[5] although not in the traffic context.

The current output interface, namely, graphs, has been a useful adjunct to the tabular output which is produced automatically by GPSS or SIMSCRIPT systems.

SUMMARY AND CONCLUSION

An important criteria in determining the usefulness of a simulator is the running efficiency. UTS–I, operating on a IBM 360/65 computer, in a 200K partition under OS/MFT–2, runs in about 5% of real time for the application described above. A large portion of this time is spent in the GPSS scheduling routine.

The problem of how to synchronize events scheduling is an especially important one in a microscopic network

simulation model. Conway[6] has presented some guidelines for choosing between the two standard methods of variable time (or next most imminent event) or fixed increment timing routines. GPSS provides only the variable timing mechanism, which may be a drawback in a traffic network simulation model, where the event densities would lead one to try fixed increment timing.

Working with G. Siegel, the author is currently implementing a mixed timing method in SIMSCRIPT–II. This will dynamically change the synchronization method from fixed increment to variable increment timing as the simulation proceeds. This should make the entire run more efficient than if either variable or fixed timing was used alone.

UTS–I was programmed in one man month after approximately 3 man months of design. These times are for the general simulator. The application was programmed using the UTS–I system in one man week after approximately three man weeks of design and one man week of data collection. The relatively short time needed to program the specific application indicates the utility of the macro approach to a traffic simulation system.

In conclusion, UTS–I, a general traffic network simulation system, composed of a set of macros and a subroutine package, has been designed and programmed. This system, and other macro systems for modeling the subsystems of our society should prove to be valuable tools in the analysis and optimization of our environment during the coming decade.

ACKNOWLEDGMENTS

The author wishes to acknowledge the assistance of M. Feldman, who with R. Atwood, S. Prensky and A. Hodges coded UTS–I as part of a Master of Engineering project in the Department of Operations Research at Cornell University. The author also wishes to acknowledge the help of G. Siegel, who is assisting the author in reprogramming UTS–I in SIMSCRIPT–II.

The UTS–I project has had the financial assistance of the Department of Transportation, Federal Highway Administration, Bureau of Public Roads under Contract No. FH–11–6913. The opinions, findings, and conclusions expressed in this publication are those of the author and not necessarily those of the Bureau of Public Roads.

REFERENCES

1 D R DREW
Traffic flow theory and control
McGraw–Hill Inc. 1968

2 A M BLUM
A general purpose digital traffic simulator
Document No 680167 Advanced Systems Development Division IBM

3 F BARNES F WAGNER JR
Case study in the application of a traffic network simulation model
Paper presented at the 34th National Meeting of the Operations Research Society of America in Philadelphia 1968

4 T SAKAI M NAGAO
Simulation of traffic flows in a network
Comm ACM 12 No 6 June 1969 pp 311–318

5 R W CONWAY
Some tactical problems in digital simulation
Mgmt Sci 10 No 1 October 1962

6 HUNTER SHERMAN J REITMAN
GPSS/360—NORDEN a partial conversational GPSS
Digest of the Second Conference on Applications of Simulation December 2–4 1968

MARSYAS—A software system for the digital simulation of physical systems

by H. TRAUBOTH

Marshall Space Flight Center
Alabama

and

N. PRASAD

Computer Applications, Incorporated
New York, New York

INTRODUCTION

New aerospace systems which are likely to be developed in the next decade such as the Space Shuttle Vehicle and Space Station will be used for more versatile missions; they will be more autonomous and independent from ground operations; and therefore, their design will be more complex than that of present space flight systems. In order to design these new systems optimally for all mission phases, extensive design analyses, evaluations, and trade-off studies have to be performed before a design can be finalized. This means that many simulations of various degrees of depth have to be run to test all possible mission conditions. Thereafter, the integrated hardware and software systems have to undergo extensive testing and checkout before they are flight ready.

For the Apollo Program, several enormous simulation facilities have been installed at contractor and NASA sites consisting of various types of simulators such as special purpose hardware simulators, flight trainers, analog, hybrid, and digital computers.[1,2] Though as early as 1955, first attempts were made to simulate analog computer block diagrams on a digital computer,[3] analog computers and special purpose simulators have played a major role in simulation until a few years ago. Since then, the great flexibility of modern digital computers has been explored in a number of developments of digital simulation languages particularly for non-real time analyses.[4] In order to direct the development of digital simulation languages, a standard language was introduced by the Sci-Committee. This language, CSSL, is particularly suitable for the simulation of analog computer-like block diagrams mixed with FORTRAN subroutines and statements.[5] Most of the common digital simulators are pre-compilers which generate FORTRAN code and order the integrator statements automatically so that the numerical integration for each integrator can be performed in the proper sequence. In comparison, the digital simulation system described in Reference 6 interprets linear block diagrams of transfer functions and converts them into a matrix equation whose coefficients are determined by the numerical convolution for each transfer function block.[7]

A blueprint of the digital simulation system being described in this paper was given in Reference 8. This simulation system addresses itself to the engineer who has little experience in simulation and in computer programming and who wants to simulate large physical systems. It could be used for a variety of applications such as for design analysis and evaluation, checkout and malfunction analysis. It could also serve as a storage and retrieval system for models—being a basis for a "model configuration control" system on a central time-shared computer. The development of models is costly, and therefore, they should be utilized by as many people as possible. The language allows a standard description of models and easy modification of already stored models, assuming the physical system is described by multiple input/output blocks or hierarchies of blocks and their interconnections using names as they appear in engineering drawings. The outputs of the simulation system are not only time-responses but also other analysis data such as stability parameters, frequency response, etc.

During the operation of a system of this magnitude

one has to reckon with certain alterations of the operational features of the system. Therefore, the simulation systems software is designed in a modular form to keep the impact of possible design changes to a minimum. To provide the additional analysis capability and to obtain efficient computation speed, analysis techniques of modern control theory have been employed for the mathematical foundation of the simulation system. The differential equations generated from the block diagram are in the form of vector–matrix state equations, which do not need to be ordered for their numerical integration.

We named this simulation system MARSYAS which stands for Marshall System for Aerospace Systems Simulation. The description of the user language, the mathematical foundations, and the software structure can be only briefly given in this paper. However, separate papers are being prepared to cover these areas more exhaustively.

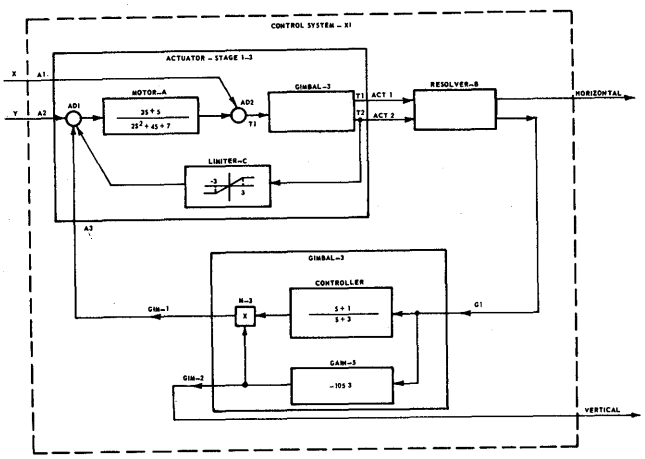


Figure 1—Example of a model described by a block diagram of multiple input/output blocks and a nested block

SIMULATION CAPABILITY

Before a physical system can be simulated, a model of its functions has to be generated. In most cases, the derivation of a model requires human judgment and, therefore, is a manual process. Only in special cases as in electrical circuits, a direct relationship exists between the physical components network and the mathematical description of its functions. In such a case the physical network can be described directly to the computer which then generates the mathematical model and solves for it.[9,10] For the design of MARSYAS, a model of the guidance and control system of the Saturn V and a model of the propulsion system of the S–IVB stage were used as test models representative for other space vehicles' electrical, mechanical, and hydraulic systems. The models referred to in this paper represent the continuous and discrete dynamics of physical systems which can be mathematically described by ordinary differential equations, algebraic equations, and logical functions.

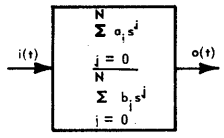
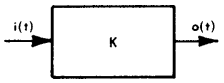
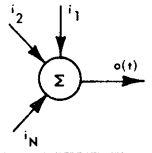
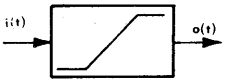
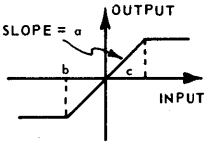
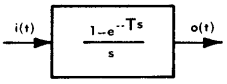
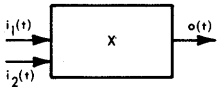
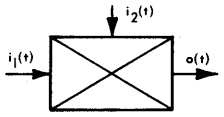
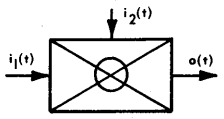
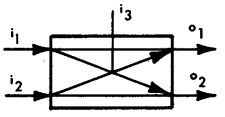
The engineer prefers to describe a model by block diagrams because their graphical representation is visually comprehensive. The blocks of the diagram can have multiple inputs and multiple outputs, and a block can contain other blocks within itself, i.e., block diagrams can be built in hierarchies, or in other words they can be nested at several levels. At the lowest level where the block cannot be broken down further, we call the block an element. We distinguish between linear and non-linear elements. A linear element is represented by a transfer function or more generally by a linear differential equation. A nonlinear element is represented by an algebraic equation or by a logical or switching function or by a nonlinear differential equation. Figure 1 depicts an example of such a model. Elements which are used frequently are called standard elements and are available in a Standard Elements List (Table I). This list is not fixed; it can be updated easily using MARSYAS statements. For infrequently used special elements, a FORTRAN subroutine can be submitted. Thus, the block diagram can contain analog computer elements, transfer functions, algebraic equations, and nonlinear ordinary differential equations.

A block diagram is specified to the computer only by the names of the blocks, inputs and outputs; by the names and values of the element parameters; and by the unidirectional interconnections of the block. The names can be up to 36 characters long so that the same names as found in engineering documentation can be used. The model thus described can be stored permanently in the Functional Data Base (FDB) or an already stored model can be modified. Only authorized personnel having the access-key can write into the FDB, whereas everybody can read out and use models of the FDB.

For a simulation run, the input signals or excitation functions into the total model can be pre-stored analytical functions such as exponential, sinusoidal, time functions or digitized signals recorded on magnetic tape. These recorded signals may be measured signals or output signals generated by a previous simulation run. The outputs of the simulation can be manifold. Any connection point in the block diagram can be chosen for obtaining a systems output signal.

The dynamic systems output signals can be plotted

TABLE I—Extract from list of standard elements

CLASS	BLOCK DIAGRAM SYMBOL	# OF INPUTS	# OF OUTPUTS	MNE-MONIC	INPUT – OUTPUT RELATION	LIST OF PARAMETERS IN THE ORDER IN WHICH THEY APPEAR IN THE ELEMENTS STATEMENT
BLOCK		1	1	BL	$\sum_{i=0}^{N} b_i \frac{d^i o(t)}{dt^i} = \sum_{i=0}^{N} a_i \frac{d^i i(t)}{dt^i}$	$N, a_N, a_{N-1}, a_{N-2}, \ldots, a_o, b_N, b_{N-1}, b_{N-2}, \ldots, b_o$
CONSTANT MULTIPLIER		1	1	CM	$p(t) = K\, i(t)$	K
ADDER		N	1	AD	$o(t) = \sum_{i=1}^{N} i_i(t)$	NONE
LIMITER		1	1	LM		a, b, c
SAMPLE AND HOLD		1	1	SH	$o(t) = i(nT), \quad nT \le t < (n+1)T,$ $n = 0, 1, 2, \ldots$	T
MULTIPLIER		2	1	ML	$o(t) = i_1(t) \cdot i_2(t)$	NONE
BOOLEAN RELAY		2	1	BR0	$o(t) = \begin{cases} i_1(t), & i_2(t) \neq 0 \\ 0, & i_2(t) = 0 \end{cases}$	NONE
		2	1	BR1	$o(t) = \begin{cases} 0, & i_2 \neq 0 \\ i_1(t), & i_2 = 0 \end{cases}$	NONE
RESOLVER		3	2	RE	$\begin{bmatrix} o_1 \\ o_2 \end{bmatrix} = \begin{bmatrix} \cos(i_3) & \sin(i_3) \\ -\sin(i_3) & \cos(i_3) \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix}$	NONE

or printed as functions of time. The steady-state response can be calculated separately in one process. For linear systems, additional analysis information can be computed such as frequency response, power spectrum, stability and parameter sensitivity.

The ability for the user to control the internal processing of the various simulation functions is kept to a minimum and restricted to functions essential to the simulation. The user can specify the relative truncation error, the integration step size, or the numerical integration method if he wants to override the standard built-in method. Algebraic loops are identified automatically by the software.

ENGINEER-ORIENTED LANGUAGE

The language is designed so that the user transmits to the computer only information which is essential to describe the model and specify the simulation run, and does not concern himself with programming the computer. If an engineer has no knowledge of modelling, he can call up a pre-stored model and run a simulation after specifying only the model input signals. On the other hand, if the engineer has FORTRAN programming knowledge, the language gives him some capability for special control of the simulation.

The MARSYAS language is divided into modules which describe independent functions of the simulation. These language modules are the:

Description Module,
Modification Module,
Simulation Module,
Continuation Module,
Post-Processing Module, and
Analysis Module.

Within these modules, several MARSYAS statements are available. Statements are written in free format and need no special ordering. However, the ordering of the modules has to follow simple logical rules, e.g., a Simulation Module has to be preceded by a Description Module because the model must be described before it can be simulated. A statement consists of an 'operator', and an 'argument field' which is composed of several arguments.

The *Description Module* is used to describe a model given in the form of a block diagram. It is headed by the operator MODEL. The ELEMENTS—statement contains the name of the element, the type of Standard Element (to be found in the List of Standard Elements), and the parameters. The parameters are written in the proper format for the particular element type and are either the numerical values or names. The numerical value of a named parameter is given by the PARAMETER—statement. If the element is not in the Standard Element List, a FORTRAN—subroutine carrying the same name as the element is given. A model stored in the Functional Data Base (FDB) is called by a SUBMODEL—statement containing the embedded model name and its input and output names. The CONNECT—statement lists strings of inputs and outputs of elements, submodels, system inputs, or system outputs to be connected to form the model block diagram. For elements or submodels having a single input/output, only the name of the element or submodel appears in the CONNECT—statement. The INPUT—statement designates names of the inputs of the model, the OUTPUT—statement designates names of the outputs of the model. The STORE—statement carrying

```
BEGIN: MARSYAS PROGRAM X.
MODEL: ACTUATOR-STAGE 1-3.
INPUTS: ACT 1, ACT 2.
ELEMENTS: BL, MOTOR - A, 2, 0, 3, 5, 2, 4, 7
        : LM, LIMITER-C, 1, -3, 3.
SUBMODEL: GIMBAL-3; INPUT: G1; OUTPUTS: GIM-1, GIM-2.
NAMING: G1, I1: GIM-1, T1: GIM2, T2.
ELEMENTS: AD, AD1, AD2.
CONNECT: A2, AD1, MOTOR-A, AD2, I1, T2, LIMITER-C, AD1
       : A1, AD2: A3, AD1
       : T1, ACT1: T2, ACT2.
END.

MODEL: CONTROL SYSTEM-X1
INPUTS: X, Y.
OUTPUTS: HORIZONTAL, VERTICAL.
ELEMENTS: RE, RESOLVER-B.

SUBMODEL: ACTUATOR-STAGE 1-3: INPUTS: A1, A2, A3: OUTPUTS: ACT 1,
          ACT 2.
SUBMODEL: GIMBAL-3; INPUT: G1; OUTPUTS: GIM-1, GIM-2
CONNECT: X, A1, ACT 1, RESOLVER-B (U1, W1), HORIZONTAL
       : Y, A2, ACT 2, RESOLVER-B (U2, W2), G1, GIM-1, A3.
       : GIM-2, VERTICAL: ACT 1, RESOLVER-B (U3).
END.

SIMULATE: CONTROL SYSTEM-X1.
INITS: ACTUATOR STAGE 1-3, MOTOR A, 1.5, 12.
EXCITE: X, FSTEP, 5.0: Y, FSIN, 1, 3000, 0.
STOP IF, TIME .GT. 2.00.
END.
PRINT: FMT, 0.01, X, Y, HORIZONTAL, VERTICAL.
FORMAT: FMT, 4F 13.8.
END.
END: MARSYAS PROGRAM X. (Figure 2)
```

Figure 2—MARSYAS-program of example in Figure 1 (It is assumed that model GIMBAL-3 is stored in the functional data base)

the proper key-code transfers the Description Module into the permanent FDB. Thus, the Description Module can be used for storing and retrieving of models as well as for describing the model for a subsequent simulation run. Figure 2 illustrates the MARSYAS—program for the example in Figure 1.

The *Modification Module* allows inserting, deleting, and disconnecting of elements and submodels through the use of the SUBSTITUTE, DELETE,—and DISCONNECT—statements. The UPDATE—statement allows additions to the Standard Elements List. Statements of the Description Module are used to specify the elements, parameters, and interconnections which are to be modified. The Modification Module can be used for modifying models of the FDB or for a subsequent simulation.

The *Simulation Module* is used to define the course of the simulation. The INITS—statement specifies initial conditions for the outputs of blocks. The EXCITE—statements tells the MARSYAS—processor what excitation functions or record tapes are fed into what system inputs of the model. If a numerical integration method other than the standard method is to be used, the INTMODE—statement specifies the integration method, and the relative truncation error or integration step-size (for fixed step-size integration methods). The STOP or HOLD statements determine the condition under which the simulation should terminate or hold, e.g., if a certain time or certain amplitudes of certain output signals have been reached. If the simulation calls for the repetition of a simulation run with modified parameters the CHANGE statement specifies these parameters and their values.

The *Continuation Module* is used to re-start a simulation run that has been terminated by a HOLD statement. This module is particularly useful when the user wishes to insert check points in a lengthy simulation run at which he can obtain intermediate outputs. (He can also change the integration mode at these check points.) Upon these outputs he can decide whether it is worth to continue the run.

In the *Post-Processing Module* the user indicates which output signals he wishes to print or plot and the format and labels of the output. The FORMAT—statement resembles the FORMAT—statement in FORTRAN.

The *Analysis Module* allows the user to designate the type of analysis output he wishes, e.g., the FREQUENCY RESPONSE—statement calls for the frequency response within the specified frequency range. Other analyses performed by the system are the determination of steady-state response, power spectrum, stability, and sensitivity for which special statements are available.

MATHEMATICAL FOUNDATION

Analytical formulation

In the formulation of the mathematical process which converts the block diagram into an internal format acceptable to the computer, we distinguish between three parts of the model: (1) the "dynamic" elements, (2) the "non-dynamic" elements, and (3) their interconnections. The output of a "dynamic" element is a function of all past input while that of a "non-dynamic" element depends only on the instantaneous input. The "constant multiplier" (or ideal amplifier) and the 'sumer' are linear "non-dynamic" elements. The linear elements 'time-delay', 'sample-and-hold', and 'differentiator' are treated as pseudo-non-linear elements. For explaining the mathematics it is assumed that the model consists of interconnected "dynamic" and "non-dynamic" elements of various types but of no nested submodels. By some software processing the MARSYAS—processor has already unwrapped these nested submodels.

The linear "dynamic" element *'transfer function'* is characterized by the following relationship:

$$\sum_{k=0}^{n} b_k \frac{d^k o(t)}{dt^k} = \sum_{k=0}^{q} a_k \frac{d^k i(t)}{dt^k} \tag{1}$$

with a_k and b_k being constant coefficients and n the order of the differential equation (or number of poles in the complex frequency domain). It is assumed that $q < n$. If $n = q$, the 'transfer function' element can simply be split into one with $a < n$ and one 'constant multiplier' and 'summer' element. $o(t)$ is the output signal and $i(t)$ the input signal of the element. Using the method as described in References 11 and 12 this differential equation can be converted into a state variable matrix equation. For the jth element we obtain

$$\dot{X}^{(j)}(t) = A^{(j)}X^{(j)}(t) + P^{(j)}i^{(j)}(t) \tag{2a}$$

$$O^{(j)}(t) = C^{(j)}x_i{}^{(j)}(t) \tag{2b}$$

$$X^{(j)}(t) = \begin{bmatrix} x_i{}^{(j)} \\ \cdot \\ \cdot \\ \cdot \\ x_n{}^{(j)} \end{bmatrix}$$

is a state vector of the jth element, and $A^{(j)}$, $C^{(j)}$, and $P^{(j)}$ are constant real matrices of the dimension $n \times n$

and can be obtained from a_k and b_k of equation (1):

$$A^{(j)} = \begin{bmatrix} \frac{b_{n-1}^{(j)}}{b_n^{(j)}} & 1 & 0\cdots 0 \\ \frac{b_{n-2}^{(j)}}{b_n^{(j)}} & 0 & 1\cdots 0 \\ \cdots\cdots\cdots\cdots & & \\ \frac{b_1^{(j)}}{b_n^{(j)}} & 0 & 0\cdots 1 \\ \frac{b_0^{(j)}}{b_n^{(j)}} & 0 & 0\cdots 0 \end{bmatrix}$$

$$P = \frac{1}{b_n^{(j)}} \begin{bmatrix} a_n^{(j)} - 1 \\ a_n^{(j)} - 2 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_0^{(j)} \end{bmatrix}$$

$$c^{(j)} = [1 \quad 0 \quad 0\cdots 0] \tag{2c}$$

For a collection of m "dynamic" elements we can write

$$\dot{X}(t) = AX(t) + PI(t) \tag{3a}$$

$$O(t) = CX(t) \tag{3b}$$

where

$$\dot{X}(t) = \begin{bmatrix} \dot{X}^{(1)}(t) \\ \cdot \\ \cdot \\ \dot{X}^{(j)}(t) \\ \cdot \\ \cdot \\ \dot{X}^{(m)}(t) \end{bmatrix},$$

$$X(t) = \begin{bmatrix} X^{(1)}(t) \\ \cdot \\ \cdot \\ X^{(j)}(t) \\ \cdot \\ \cdot \\ X^{(m)}(t) \end{bmatrix},$$

$$A = \begin{bmatrix} A^{(1)} & & & & \\ & \cdot & & & \\ & & \cdot & & \\ & & & A^{(j)} & \\ & & & & \cdot \\ & & & & & \cdot \\ & & & & & & A^{(m)} \end{bmatrix},$$

$$P = \begin{bmatrix} P^{(1)} & & & \\ & \cdot & & \\ & & \cdot & \\ & & & P^{(m)} \end{bmatrix},$$

$$C = \begin{bmatrix} C^{(1)} & & & \\ & \cdot & & \\ & & \cdot & \\ & & & C^{(m)} \end{bmatrix},$$

$$I(t) = \begin{bmatrix} i^{(1)}(t) \\ \cdot \\ \cdot \\ \cdot \\ i^{(m)}(t) \end{bmatrix},$$

and

$$O(t) = \begin{bmatrix} O^{(1)}(t) \\ \cdot \\ \cdot \\ \cdot \\ O^{(m)}(t) \end{bmatrix}$$

We now assume that the "dynamic elements are con-

nected in any way through 'constant multipliers' and 'summers' to form a linear model. We then can write the following *linear interconnection matrix equations*:

$$I(t) = EO(t) + FU(t) \tag{4a}$$

$$W(t) = GO(t) + HU(t) \tag{4b}$$

In the above

$U(t)$ = vector of inputs (excitations) into model

$W(t)$ = vector of outputs from model

m = number of "dynamic" elements in model

k = number of inputs into model

l = number of outputs from model

and E, F, G *and* H are matrices having appropriate dimensions.

The coefficient e_{ij} in E means the total constant gain along the path from the output o_j of "dynamic" element j to the input i_i of "dynamic" element i. The coefficient f_{ij} in F is the total constant gain along the path from the model input u_j to the input o_i. The coefficient g_{ij} in G is the total constant gain from the output o_j to the model output w_i. The coefficient h_{ij} in H is the total constant gain from the model input u_j to model output w_i.

There may be linear systems that do not have the interconnection equations (4a and 4b) but wherever possible the MARSYAS processor generates these interconnection equations.

By substituting equation (4a) into equation (3a) and equation (3b) into equation (4b) we obtain the model overall matrix equations:

$$\dot{X}(t) = A^*X(t) + P^*U(t) \tag{5a}$$

$$W(t) = C^*X(t) + D^*U(t) \tag{5b}$$

where

$$A^* = A + PEC, \quad P^* = PF$$

$$C^* = GC, \quad \text{and} \quad D^* = H. \tag{5c}$$

We now include *nonlinear* elements of the form

$$y^{(j)}(t) = f^{(j)}(r^{(j)}(t), t) \tag{6}$$

where $y^{(j)}(t)$ denotes the output vector, $r^{(j)}(t)$ the input vector, and $f^{(j)}$ the input–output function associated with the jth nonlinear element. For a collection of nonlinear elements it is

$$Y(t) = \bar{F}(R(t)). \tag{6a}$$

For the inputs to all "dynamic" elements and to all nonlinear elements, respectively, we write the following *nonlinear interconnection matrix equations*

$$I(t) = EO(t) + FU(t) + KY(t) \tag{7a}$$

$$R(t) = E'O(t) + F'U(t) + K'Y(t) \tag{7b}$$

where the vectors $Y(t)$ and $R(t)$ represents the collection of the output vectors and input vectors of all nonlinear elements of the model. The output vector $W(t)$ for the model becomes

$$W(t) = GO(t) + HU(t) + K''Y(t) \tag{7c}$$

The matrices $E\cdots E''$, $F\cdots F''$, and $K\cdots K''$ represent the cumulative gain along the various paths between the inputs and outputs of the "dynamic" elements, nonlinear elements in the model.

By substituting equation (7a) into equation (3a) and equation (3b) into equation (7c) and using equation (6) and (7b) we obtain

$$\dot{X}(t) = A^*X(t) + P^*U(t) + N(O, U, t) \tag{8a}$$

and

$$W(t) = C^*X(t) + D^*U(t) + M(O, U, t) \tag{8b}$$

with A^*, P^*, C^* and D^* being the same matrices as in equation (5c). $N(O, U, t)$ and $M(O, U, t)$ are the nonlinear column vectors $(n_1(O, U, t), n_2(O, U, t), \cdots, n_m(O, U, t)$ and $(m_1(O, U, t), m_2(O, U, t), \cdots, m_l(O, U, t)$ respectively. It is assumed here that there are no "algebraic loops" in the model. An overview diagram of the mathematical process is given in Figure 3.

The matrices A^*, P^*, C^*, and D^* are characteristics for a linear model and they can be used for a number of analyses.

The *stability* of the system may be determined from a knowledge of the eigenvalues of matrix A^*, i.e., all

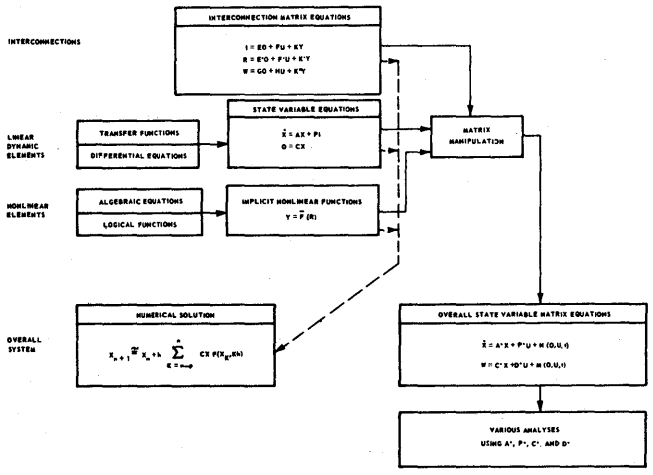


Figure 3—Overview diagram of the mathematical process

eigenvalues must lie in the left half of the complex frequency plane for stability. The frequency response of the system may be obtained, if it exists, from the Fourier Transform of the impulse response. The impulse response is obtained numerically as the solution of the autonomous system with appropriate initial conditions as follows:

$$x(t) = e^{A^*t}x(0-) + \int_{0-}^{t} e^{A^*(t-\tau)}P^*\Delta \, d\tau$$

where

$$\Delta = [\delta_{ri}(\tau) \cdots \delta_{ri}(\tau)]^t$$

and

$$\delta_{ki}(\tau) = \begin{cases} 0, & k \neq i \\ \delta(\tau), & k = i \end{cases}$$

and $\delta(\tau)$ is the unit impulse function. Hence $x(t)|_{t=0^+} = e^{A^*t}P^*e$ where $e = [00\,..\,1\,,\,,\,0]^T$ and the unity element in the e vector occurs in the ith component. Noting that $e^{A^*t}|_{t=0} = I$, the identity matrix, we obtain

$$x(t) = e^{A^*t}P^*e.$$

The sensitivity function relating the instantaneous rate of change of a system state or output with respect to a parameter is obtained as the solution of an auxiliary linear differential equation referred to as the sensitivity equation. Denoting by q the parameter of interest, we have

$$\frac{d}{dt}\frac{\partial x}{\partial q} = A^*\frac{\partial x}{\partial q} + \frac{\partial A^*}{\partial q}x + \frac{\partial P^*}{\partial q}U$$

subject to the initial conditions

$$\left.\frac{\partial x}{\partial q}\right|_{t_0} = \frac{\partial x_0}{\partial q}$$

Numerical solution

Nearly all numerical methods for the solution of differential equations are based on the numerical integration of first order differential equations (16), (13) and (14). Hence, the state-variable matrix equations are particularly suited for these methods. A system of first order differential equations, in general of the form

$$\dot{X}(t) = F(X(t), t), \tag{13}$$

is usually approximated by single-step evaluation or multi-step predictor-corrector methods where $\dot{X}(t)$, $X(t)$, and $F(X(t), t)$ are the column vectors $[\dot{X}_1(t), \cdots, \dot{X}_m(t)]^T$, $[X_1(t), \cdots, X_m(t)]^T$, and $[f_1(X, t), \cdots, (M(X, t)]^T$, with M equal to the total number of state variables within the model. The most common *single-step method* is the Runge-Kutta (4th order) which approximates equation (13) for $t = (n + 1)h$ into

$$X_{n+1} \cong X_n + 1/6(K_1 + 2K_2 + 2K_3 + K_4) \tag{14}$$

with the vectors

$$K_1 = hF(X_n, t_n),$$

$$K_2 = hF(X_n + h/2, t_n + K_1/2),$$

$$K_3 = hF(X_n + h/2, t_n + K_2/2),$$

$$K_4 = hF(X_n + h, t_n + K_3),$$

where h is the time step.

The multi-step predictor-corrector methods are of the form

Predictor:

$$X^P((n + 1)h) \cong X(nh) + h\sum_{k=n-p}^{n} c_kF(X(Kh), Kh) \tag{15}$$

Corrector:

$$X((n + 1)h) \cong X(nh) + h\sum_{k=n-q}^{n} d_kF(X(Kh), Kh) + hd_{n+1}F(X^P((n + 1)h), (n + 1)h) \tag{16}$$

with p being the order of the predictor and q being the order of the corrector polynomial; and c_k and d_k are constant coefficients depending on the method used. For instance, for the Adams-Bashforth predictor it is

$$c_{n-3} = -9/24,$$

$$c_{n-2} = 37/24,$$

$$c_{n-1} = -59/24,$$

$$c_n = 55/24,$$

and for the Adams-Moulton corrector it is

$$d_{n-2} = 1/24,$$

$$d_{n-1} = -5/24,$$

$$d_n = -19/24,$$

$$d_{n+1} = 9/24 \tag{15}$$

One can show that numerically solving the overall matrix equations (7) and (8) is not the most efficient way, because the matrices A^*, P^*, C^*, and D^* contain many zero elements. Less computation steps are necessary if one uses the individual equations (3), (6), and (7).

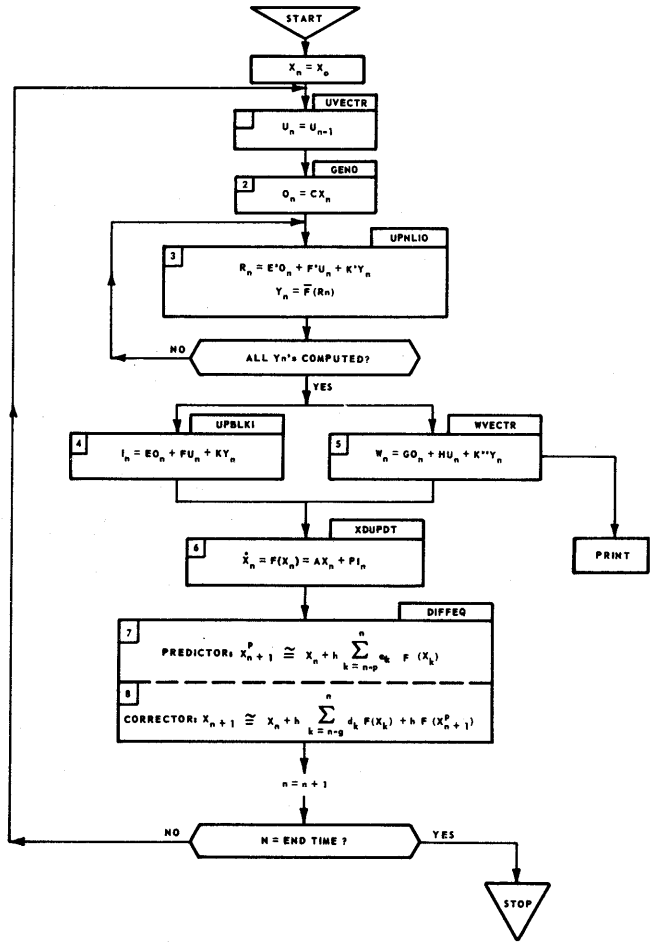


Figure 4—Flow of numerical solution of matrix equations

The numerical process is the following (see Figure 4):

1. We first update the excitation vector $U(t)$ for $t = nh$ and calculate

$$O(nh) = CX(nh)$$

2. or

$$O_n = CX_n \tag{17}$$

From now on, we will use the subscript n for the time instant $X = (nh)$.

3. Then, part of the input vector $R(t)$ is computed from equation (7b) for those nonlinear elements whose input is not connected to other nonlinear elements. For these nonlinear elements, the output can now be calculated using equation (6). Then, that part of $R(t)$ can be calculated which contains known $y_n^{(j)}$. Through alternate use of equation (7b) and (6) and assuming that the nonlinear equations were already properly ordered the complete vector Y_n can be calculated.

Now, the input vector I_n can be obtained by use of equation (7a).

4. $I_n = EO_n + FU_n + KY_n$ (18)

and the model output vector w_n of equation (7c) is

5. $W_n = GO_n + HU_n + K''Y_n.$ (19)

Knowing I_n we go to equation (3a)

$$\dot{X}(t) = F(X(t), t) = AX(t) + PI(t) \tag{3a}$$

and evaluate

6. $\dot{X}_n = F_n = AX_n + PI_n.$ (20)

Then using equation (15) the predicted value of X_{n+1} is

7. $$X_{n+1}^p \cong X_n + h \sum_{k=n-p}^{n} c_k F(X_k) \tag{21}$$

and the corrected value of x_{n+1}, i.e., the final value of X, for $t = (n+1)h$ is

$$X_{n+1} \cong X_n + h \sum_{k=n-a}^{n} d_k F(X_k) + hF(X_{n+1}^p) \tag{22}$$

In Figure 4, the sequence of the numerical evaluation is outlined. One can see that within each block only matrix multiplications and additions have to be performed. The row-by-column multiplications are not dependent on each other; hence, the sequence in which they are executed is immaterial.

This property has the advantage that several vector multiplications and additions could be computed simultaneously resulting in a tremendous speed-up of the computations if several parallel processors were available.

SOFTWARE STRUCTURE

General

The prime objective of the MARSYAS software is to transform a MARSYAS program, which describes a model and specifies the simulation, into a FORTRAN program that contains the arrays and subroutines for the numerical solution of the various matrix equations. The MARSYAS software is a precompiler for compiling MARSYAS language statements into a set of FORTRAN subroutines, arrays, and control cards, i.e., the Object Program, and a controller for the execution of these FORTRAN programs. It is built for time-sharing operation on MSFC's central computing facility, the UNIVAC 1108. The software allows several

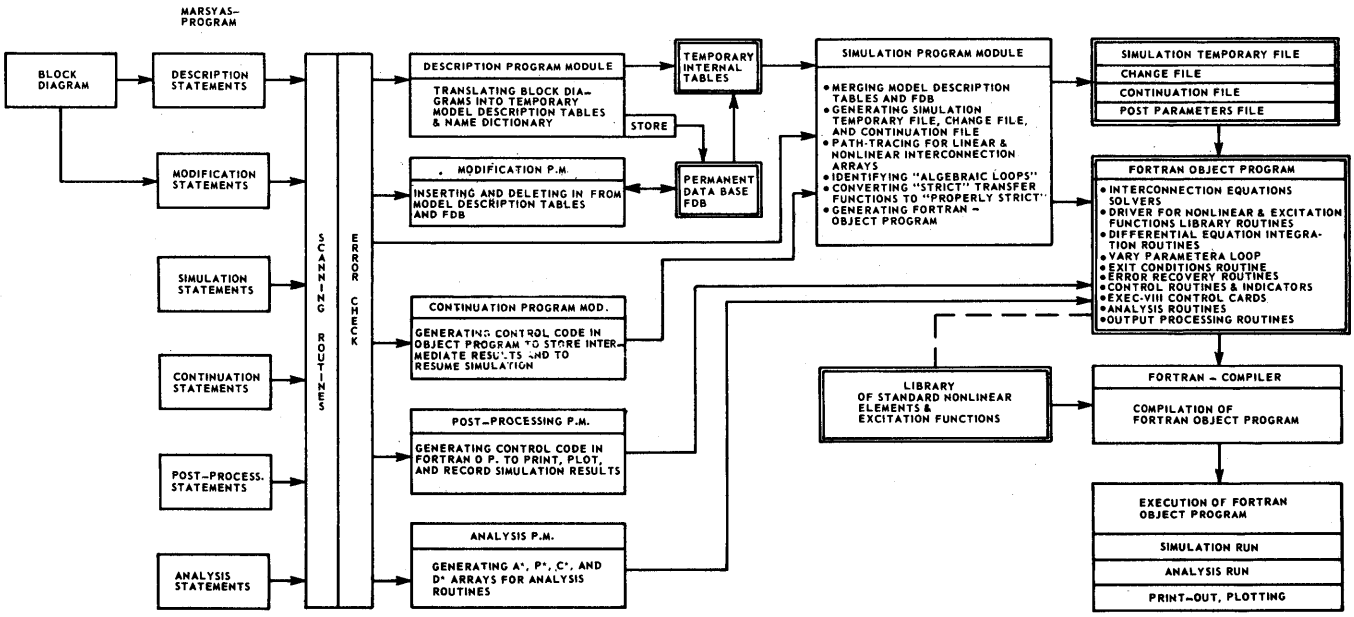


Figure 5—Overview of MARSYAS systems software

users to access MARSYAS and its models simultaneously from remote stations. Although maximum usage of the UNIVAC 1108-EXEC VIII operating system is made, the MARSYAS software could be converted for other facilities since it is written in FORTRAN. In order to have the MARSYAS system evolve while it is in operation it is mandatory to break it into many relatively independent program modules. The major Program Modules (PM) are defined by the language modules. Hence, we distinguish between the Description PM, modification PM, Simulation PM, Continuation PM, Post-Processing PM, and Analysis PM. Outside these Program Modules there are other programs, such as the FORTRAN Object Program (OP), library routines for 'standard elements' and 'excitation functions', scientific subroutines for the numerical integration of first order differential equations, scanning routines, error recovery routines, and control routines.

The Object Program is compiled by the FORTRAN-compiler and then executed like any manually-generated FORTRAN program. Figure 5 is an attempt to give an overview of the main flow of action; it is of course, a simplification and does not stand a precise judgement.

Object program

The FORTRAN Object Program (OP) consists of the MAIN—program, which has a fixed structure, and which calls up various subroutines of fixed and variant structure. "Variant" means that the program statements vary with each simulation run. The OP reads from four files:

1. The Simulation Temporary File, which contains all arrays for solving the matrix equations of Figure 4.
2. The Change File, which contains those parameters which are to be changed, as specified by the CHANGE—statement.
3. The Continuation Permanent File, which stores intermediate results of a discontinued simulation run, such as the state space vector $X(t)$.
4. The Post Parameters File, which keeps parameters and labels for printing and plotting.

The OP generates the Simulation Output Tape, which contains the numerical values of all model output signals for a complete simulation run.

The subroutines of fixed structure perform the following functions:

XDUPDT	evaluates the derivative $\dot{X}_n = F(X_n)$ of equation (20).
PCHNG	changes parameters in accordance with the Change File.
TIDY	writes intermediate simulation results into the Continuation Permanent File.
GENO	generates the block output vector O_n from equation (17).

The variant subroutines are the following:

UVECTR	generates the vector U_n by accessing the library routines for the various excitation functions.
WVECTR	generates the vector W_n from equation (19).
UPGAIN	computes the array $G(I)$ which contains the cumulative gain of the Ith path between a "source" (variable I, W, or Y) and a "terminator" (variable O, U, or R).
UPNLIO	updates outputs of nonlinear devices, which are already sequenced in the proper order for evaluation, using library routines for the various standard nonlinear elements.
UPBLKI	generates the vector I_n from equation (18).
DIFFEQ	is the general integration routine which calls a specific integration routine such as RKG for Runge-Kutta-Gill (4th order) or AM for Adams-Bashforth-Moulton predictor–corrector, etc.
POST	writes the Simulation Output File.
EDIT	prints output data generated by POST.
PLOT	plots output data generated by POST.
PDP	Elements specify the dimensions of the various COMMON arrays in the OP routines.
ANALYSIS	is a collection of special subroutines to generate the overall matrices A^*, P^*, C^*, and D^* and to perform special analysis computations.

Description and modification program module

The subroutines of the Description Module translate the MARSYAS statements describing a model block diagram into the Model Tables for the elements, inputs, outputs, parameters, and connections. These tables are packed to save storage space. Program names are converted into identification code words (ID) via the Name Dictionary, so that in the internal processing the shorter Name ID's can be used. The connections are ordered into pairwise connections (predecessor/successor pairs). Since the statements can be written in any order, the END—routine has to check for certain formal errors such as undefined names, missing elements, and improper connections, after all tables have been filled.

If the model is to be stored into the permanent Functional Data Base (FDB) the temporary Model Tables are transcribed into the FDB—file.

The Modification Program Module subroutines are similar to those of the Description Module in the sense that they also access the Model Tables and modify them.

Simulation and continuation program module

The subroutines of the Simulation Program Module have to perform a variety of functions. The tables of the Functional Data Base for models specified in the SUBMODEL—statements, and the temporary Model Tables, are merged into the Model Tables File (MTF). Blocks representing transfer functions of equal numerator and denominator order are converted into blocks where the order of the numerator is one less than that of the denominator. The Simulation Module statements for initial conditions, excitations, integration mode, etc., are translated into the Simulation Specification Tables. The Connection Tables of the MTF are used for path tracing to generate the Gain Table, which contains the cumulative gains between the various terminals. The nonlinear elements are sequenced in the Nonlinear Elements Table. From various intermediate tables such as the Model Tables Files, Simulation Specifications File, etc., the final files used by the Object Program, i.e., the Simulation Temporary File, Change File, and Continuation Permanent File; and the Program File (i.e., the Object Program) are generated.

The Continuation Program Module accesses the Continuation Permanent File and places among other control data the X-vector into the Initial Conditions Record of the Simulation Temporary File, so that the simulation run can be continued by the Object Program.

Post-processing and analysis program module

The Post-Processing Program Module generates the Post Parameters File and the subroutines POST and EDIT of the Object Program. The Analysis Program Module generates the parameter arrays representing the matrices A^*, P^*, C*, and D^* for the various analysis subroutines in the OP.

Potentials and Implementation of MARSYAS

The scientific subroutines of the Object Program and the library subroutines have been successfully tested with linear and nonlinear test cases. The systems software is coded and is presently in the final checkout process on MSFC's time-sharing computer UNIVAC 1108. The detailed design specifications are documented and revised.[17] The present implementation, however,

does not include blocks of differential equations of arbitrary format and time varying systems. The systems software for the various analyses has not been implemented yet.

The mathematical foundation and the software structure allow for expanding the capability of MARSYAS and for improving its language and internal processing. Thus, elements in the model block diagram which contain ordinary differential equations of any order in form of mathematical equations will be included. The coefficients of the differential equations can also be functions of time. While presently "algebraic loops" can be only identified but not solved for (except by inserting an artificial time delay or an implicit function) it is expected to include separate mathematical procedures which assure convergence wherever possible to solve identified algebraic loops. The matrix equation formulation of the physical system in MARSYAS resembles closely to the way electrical networks are mathematically described for analyzing them on the digital computer.[10,18] Methods for partitioning sparse matrices in electrical circuit analysis might, therefore, be applicable to further improve the computation speed particularly for large physical systems.[19] The language is structured in such a way that it should be straightforward to input block diagrams and simulation statements via graphical display into the computer and thereby enhance the man-machine communications tremendously. The MARSYAS—language improvements will be stalled until the user has gained practical experience in the operation of MARSYAS.

As was pointed out previously, the MARSYAS—language is well-suited for an easy description of dynamic models, and the MARSYAS—software system allows readily to store, retrieve, and modify models in a central data bank. Thus, MARSYAS could become the basis for a "Model Configuration Control" System which keeps the information about the functions of aerospace hardware up-to-date and available to many engineers and systems analysts in a common language, similar as to the computerized Vehicle Configuration Control System which keeps the information about the configuration of the hardware up-to-date. The easiness and speed of setting up and running a precisely repeatable simulation together with its special analysis capability make MARSYAS a valuable tool for analyzing and evaluating complex aerospace systems.

ACKNOWLEDGMENT

This project would not have progressed as much without the great endeavor and cooperation of all team members. Special credit is given to H. Wisnia, H. Gabow, H. Smith and J. Reiss of Computer Applications Incorporated, who made major contributions in the language implementation and software design, and to Dr. R. Sevigny and T. Ballentine of Computer Science Corporation who are responsible for the software implementation and programming.

REFERENCES

1 G H GALE
The Boeing Huntsville simulation center
Simulation Vol 5 No 4 October 1965

1a *Saturn V system development breadboard facility data plan*
Boeing Corporation Document No D5-15207 NASA Contract NAS8-5608

2 *Flight software development laboratory*
IBM Document No IBM-68-U60-0022 under NASA Contract 1968

3 R D BRENNAN R N LINEBARGER
A survey of digital simulation: digital analog simulator programs
Simulation Vol 3 No 6 December 1964

4 J C STRAUSS
Digital simulation for continuous dynamic systems—An overview
Proceedings of Fall Joint Computer Conference 1968

5 *The SCi continuous system simulation language (CSSL)*
SCI-Committee
Simulation Vol 9 No 6 December 1967

6 H H TRAUBOTH
Digital simulation of general control systems
Simulation June 1967

7 H H TRAUBOTH
Recursive formulas for the evaluation of the convolution integral
Journal of ACM Vol 16 No 1 January 1969

8 H H TRAUBOTH J R MITCHELL J W MOORE
Digital simulation of an aerospace vehicle
Proceedings of ACM National Meeting Washington August 1967

9 *1620 electronic circuit analysis program (ECAP)*
Application Program 1620-EE-02X IBM Corporation Data Processing Division White Plains New York 1964

10 *NASAP 69/I network analysis for systems applications program*
Contract NAS12-663 NASA Electronic Research Center January 1969

11 L A ZADEH
Linear system theory
McGraw-Hill New York 1963

12 P M DERUSSO et al
State variables for engineers
John Wiley New York 1965

13 P R BENYON
A review of numerical methods for digital simulation
Simulation Vol 11 No 5 November 1968

14 H R MARTENS
A comparative study of digital integration methods
Simulation February 1969

15 S D CONTE
Elementary numerical analysis
McGraw-Hill New York 1965

16 P HENRICI
Discrete variable methods in ordinary differential equations
John Wiley & Sons New York 1962

17 *Programming specifications for the MARSYAS system Vol I (1968) Vol II and III (1969)*

Computer Applications, Inc New York NASA Contract NAS8–21061
18 F H BRANIN
Computer methods of network analysis
Proceedings of the IEEE Vol 55 No 11 November 1967
19 W F TINNEY J W WALKER
Direct solutions of sparse network equations by optimally ordered triangular factorization
Proceedings of the IEEE Vol 55 No 11 November 1967
20 *Requirement specification part I and analysis of dynamic simulation methods for launch vehicle component level simulation*
Apollo Support Department General Electric Company Daytona Beach Florida
Final Report March 1966 MSFC Contract NAS8–20060 mod 1

Remote real-time simulation

by OMRI SERLIN

Control Data Corporation
Sunnyvale, California

and

ROBERT C. GERARD

The Boeing Company
Seattle, Washington

INTRODUCTION

Real-time simulation has traditionally been associated with small, committed digital computers located in close proximity to analog computers or simulation equipment. However, the recent development of multiprogrammed hybrid systems[1] showed clearly that exclusive control of the digital computer by one simulation job at a time is not only unnecessary, but is, in fact, economically indefensible, due to the abnormally high proportion of idle time inherent in simulation work. Users of such multiprogrammed systems have been operating successfully, two or more at a time, with elementary teletype or CRT-keyboard terminals as their only means of communication with the central computer. It is, therefore, reasonable to assume that, given the appropriate software and hardware elements, such terminals could be located substantial distances away from the central computer without compromising the level of interactive control of the simulation.

To test the validity of this assumption, an experiment has been carried out at the Boeing Company's facilities in Seattle, using the Control Data®* 6600 and 1700 computers. In the experiment, a real-time simulation of a supersonic aircraft is performed. A simplified "cockpit" and analog recording equipment are located some 10 air-miles away from the 6600, where the mathematical model of the aircraft is implemented. This experiment supplied the motivation for, and some of the results reported in this work. Many of the considerations, however, are universal in nature. In addition, the paper speculates on the desirable software and hardware features that will permit even higher levels of efficiency in running remote real-time simulations.

*Registered Trademark of Control Data Corporation

WHY REMOTE SIMULATION?

Many industrial organizations, government installations and universities possess large-scale computer installations to support their scientific batch and/or time-sharing needs. Remotely located simulation facilities can tap the power of these fast, sophisticated processors at a fraction of the cost of procuring and installing committed systems for specific needs. The central computer is likely to be fast enough so that the simulation can be coded in FORTRAN; the resulting advantages include faster debugging and easier modifications to accommodate model growth and change. The removal of the analog-digital proximity constraint permits better matching of sites to the differing hardware requirements of simulation gear and computer mainframe and peripherals.

PECULIARITIES OF REAL-TIME SIMULATION

Real-time simulation is characterized by computing tasks that are prompted by *external events* ("interrupt signals"), and that require the *results* of the computation to be made available to the external equipment either at the time of another external event or a specified time period after the occurrence of the initiating event. The external events can be periodic or non-periodic. In the latter case they are termed "asynchronous interrupts". Typical periods ("frame times") are of the order of 10 to 100 milliseconds. The amount of com-

puting time required in every such frame depends entirely on the complexity of the mathematical model and the speed of the processor. Using the CDC 6000 as an example, the required compute time for a typical aerospace simulation varies between 10% and 60% of the frame time.

These timing constraints distinguish the real-time simulation from time-sharing (TS) or remote-job-entry (RJE) systems where the constraints are one or two orders of magnitude less severe. Even more important is the fact that failure to meet a timing constraint is merely a nuisance in a TS or RJE environment, whereas the same effect in real-time simulation is often fatal in that it can cause irreparable damage to the simulation. (This is less true in pure digital simulations than in hybrid ones.) Tasks characterized by such fatal outcome of timing errors are often termed "time critical" or "constrained" computation.

The amount of data exchanged between the computer and the simulation equipment depends primarily on the division of tasks between these components of the system. Representative figures might be 10 to 25 inputs and 10 to 50 outputs at analog (~15 bit) precision. Discrete (on–off) signals as well as puredigital quantities invarying sizes and numbers could also be exchanged periodically.

WHAT ARE THE UNKNOWNS?

By "remote" we mean that such simulation hardware as analog computers, AD conversion gear, cockpit mockups, and so forth, are located so far from the central computer, where the mathematical model of the aircraft or spacecraft is implemented, that standard cables cannot be used and some sort of distant communications equipment is required to carry signals between the computer and the remote facility.

The success of the remote simulation concept depends on several fundamental issues:

a. *The Communications Facility*

Standard switched or private line wide-band offerings of the common carriers, as well as private optical or cable communications links, are available. The *reliability* of these services is of interest. A related question is that of *error handling*, i.e., the recovery algorithm.

Of crucial importance (as will be seen later) is the *speed* of the communications link. In this respect, the question divides into two parts. First, how adequate is the available service? Secondly, how fast *should* it be?

b. *The Remote Terminal*

Since the central computer exchanges signals with the local communications interface only, it is completely unaware of the type and organization of the equipment at the remote site. Thus, these parameters are flexible and the designer can choose any one of several possible configurations. The question is, therefore, what is the best remote terminal configuration, and what criteria should be used to measure its adequacy?

c. *Effect on Central Site*

One of the basic assumptions of this work is that the primary mission of the central site is *other* than supporting real-time simulation. It is then extremely important to establish (in advance if possible) the effects of adding the simulation to the standard load, which is presumed to be a combination of batch, time-sharing and remote-job-entry activities.

THE EXPERIMENTAL SYSTEM

Figure 1 is a schematic diagram of the system which was used to study some of the questions just posed. The following is a brief description of the components of the system.

a. *The Communication Components*

The wide-band *communication link*, provided by the telephone companies, is classified according to its band-

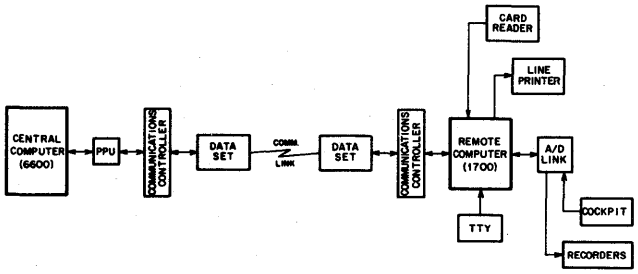


Figure 1—The experimental system

The experimental system consists of the CDC 6600 "supercomputer," standard telephone company wide-band communications link operating at 40.8 kilobits/sec., the 1700 control computer, the CDC 1500 Analog-Digital Interface, and Boeing-supplied cockpit and analog recorders. The simulation complex (1700, 1500, and simulation gear) is located some 10 air-miles away from the 6600.

width. The most common wide-band services are the 48 KHz and the 240 KHz lines. The bandwidth places an upper limit on the data rate that can be transmitted over a given line. However, the actual transmission rate is determined by the *data set.*

Data sets are required at each end of the communication link. A *data set* (also known as modulator-demodulator or "modem" for short) is a special purpose, bi-directional analog-digital converter. Serial binary information, presented to the data set by the computer, control the modulation pattern of a continuous carrier signal* which is fed into the communication link. At the receiving end, the modulated analog wave shape is decoded into a series of on–off signals which are outputted from the data set serially. Data sets for wide-band service (about 20,000 bits/sec and over) are generally clocked. The clock rate controls the transmission speed. The three most common speeds are 40.8, 50, and 230.4 kilobits/sec.

The data set does not attempt any interpretation of the data passing through it. The *Communications Controller,* which interfaces the data set to the I/O facilities of the computer, is responsible for decoding the incoming data. In order to provide some facility for exchanging control information between the remote and local equipment, the controller can generally be expected to establish at least two modes. An IDLE mode indicates that no data is being received. In this mode special decoding circuits can search the incoming bit pattern for any number of special codes. The two most basic codes required are: (a) a code to signify the beginning of a data exchange, and (b) a code to signify that the remote station desires attention. The latter code can be wired to trigger an interrupt in the computer or a special status bit in the controller.

The data exchange is terminated when the computer simply does not accept (or supply) a word within the time required to receive (or transmit) the previous word. This, and several other features of the communications controller, are designed to achieve high-speed, efficient transfer of block data. The software implications of this organization are examined later.

The specific equipment used in our experiment included a 48 KHz line and a Bell®** 303 data set, clocked at 40.8 kilobits/sec.

b. *The Computer Facilities*

The central computer complex at which the work was done consists of two CDC 6600 computers, each with 131K (60 bit) words of memory. Briefly, the 6600 is an extremely powerful, large scale computer, incorporating a very fast 60-bit central processor and 10 peripheral processors, each with a private 4K 12-bit memory and a repertoire of simple arithmetic and I/O instructions. The peripheral equipment includes disks, tapes, graphic terminals, interactive typewriter terminals, and unit record equipment. Each 6600 is also equipped with a 4-port high speed communications controller, but normally only two RJE terminals are connected to a mainframe at a time. The AD link and simulation gear were installed at one such RJE terminal, consisting of a CDC 1700 with 16K of 1 microsecond, 16-bit memory, a disk operating system, a high speed communications controller and unit record equipment. More detailed information on these computers can be found in References 2 and 3.

c. *Modification of Facilities*

Certain revisions to the normal mode of operation had to be made so that testing of the remote real-time simulation concept could be carried out. We had determined that a realistic minimum frame time for RTS would be about 20 milliseconds. We also determined that the standard Export/Import (RJE) package could not be used to support the simulation. We, therefore, dedicated the high speed multiplexer (communications controller) and a peripheral processor to the real-time application. This meant that under operational conditions all other remote job entry work would have to be handled by the second 6600, or additional hardware would have to be purchased to allow RTS and RJE to run simultaneously on the same processor.

It was also determined that the real-time job should not be subject to storage relocation and, indeed, should be able to interrupt the storage relocation task in order to direct the CPU to the time critical application. This resulted in reducing the priority of the storage relocation program and assigning the highest CPU priority to the real-time job. It also resulted in assigning the real-time job to a memory area adjacent to the fixed-size system tables. This had the effect of automatically eliminating the need to relocate this job, without any modifications to the system.

Except for these modifications, which were of a minor nature, we were able to implement the real-time simulation within the framework of the current Boeing operating system. The ease with which we achieved this conversion is due to a large extent to the unique organization of the CDC 6000 computers, where the CPU is normally completely divorced from handling any I/O tasks. In a more conventionally-organized

*Completely digital transmission systems are available in some areas.

**Registered Trademark of AT&T.

TABLE I—Analog Word Transmission Rates (Words/Sec.) at Various Line Speeds

Bits/Sec.	12 bits/word	15 bits/word
40.8 K	3400	2720
50 K	4166	3333
230.4 K	19,200	15,360

computer, the real-time job will have to share the CPU with time-critical I/O operations (e.g., for avoiding lost data conditions on fast devices), and hence with non-interruptable system routines. In such an environment the conversion may be considerably more difficult.

d. *The Linkage and Simulation Gear*

In an attempt to keep the demonstration as simple as possible, while maintaining a degree of realism in the simulation, we chose to display outputs on strip chart recording equipment and to limit inputs to pitch and roll control from a simple "joy stick". Control Data provided a 1500-type interface between the 1700 remote terminal and the control and recording devices. The interface equipment included 16 AD and 16 DA channels with a resolution of 15-bits (including sign) and a maximum range of ±128 volts.

e. *The Applications Program*

The mathematical model program that was implemented on the 6600 was a fairly simple six degree of freedom rigid body representation of a supersonic aircraft in a high speed cruise condition. Debug and checkout time had to be held to a minimum, so the voluminous data normally used to describe engine and aerodynamic characteristics was eliminated, as was the autopilot. The simulation did include a complete set of equations of motion, linear aerodynamic coefficient buildup, all coordinate transformations, a nonlinear atmosphere, effects of wind and a complete set of flight control equations, including both longitudinal and lateral stability augmentation systems. Manual inputs were pitch and roll control. (We assumed fixed throttle and trim settings.) The outputs included all data normally displayed on the pilot's flight instrument panel. The simulation was quite typical of those used for piloted studies of airplane handling qualities, stability and control problems, or flight control system optimization.

The simulation along with the central control program (to be described shortly), the associated service routines (all coded in FORTRAN IV), system routines and I/O buffers required less than 12K of central memory. Execution time per frame was approximately 4 milliseconds.

SPEED AND RELIABILITY OF THE COMMUNICATION LINK

As indicated previously, standard wide-band facilities permit operation at 40.8, 50, and 230.4 kilobits/sec. Table I below translates these speeds to words/sec., for 12 and 15 bits per word. Maximum packing (into data-channel-width words) is assumed and incompletely packed words, as well as header (Begin Data) and trailer (cyclic code) words appended by the controller, are ignored. Hence actual rates can be expected to be slightly lower than those indicated in Table I.

State-of-the-art ADC's convert at rates of between 50,000 and 250,000 15 bit words/sec. It is clear that the time to transmit a block of AD data over the communication link, even at the highest available speed, is much more significant than the time required to make the conversion. Since the computation at the central computer cannot begin until the AD data is in, and must be completed sufficiently ahead of the next frame, to allow time for the transmission of DA data, the speed of the communication link is a crucial factor in determining the permissible quantities of AD/DA data for any given frame time, or, conversely, the possible frame time for a given amount of data.

It is important to place this restriction in the proper perspective. In our specific case, the 2 AD, 14 DA and 10 control words used required about 8 milliseconds to transmit over the 40.8 kilobit/sec. line. Even allowing for some growth in the complexity of the central program, there is still enough time within the 20 millisecond frame to exchange some 20 additional quantities per frame. With the overlap option, which is described later, essentially *all* of the frame time could be used for transmission. Furthermore, a committed computer, comparable in cost to our remote terminal, would have possessed a much slower CPU and therefore provide substantially reduced performance in terms of the permissible computation per frame, or the permissible minimum frame time.

An interesting development recently reported[4] is that of laser communication links offering speeds of up to 250,000 bits/sec. These devices have been rumored to be commercially available at a very attractive purchase price. They require line-of-sight clear path, of course; their range is quite limited; and their performance in adverse weather conditions is, apparently, as yet unproven. Nevertheless they may be attractive in some

situations. Microwave facilities as well as privately constructed cable links are also of interest, as are new common carriers' offerings of, for example, a 50 kilobits/sec. *switched* (dial-up) service and the promise of megabits/sec. links as a fall-out from the development of the phone-TV service (where you see as well as hear the person you call).

The question of speed is quite fundamental. If line speeds were compatible to the data rates sustained by local equipment, it would be possible (at least theoretically), through proper design of the communication equipment, to make the remote equipment appear to the computer identical to local equipment. The computer would then be able to function the remote equipment and interrogate its status, and the remote equipment would be able to send many varied interrupt requests to the computer. The cost of very-wide-band lines is likely to prevent this possibility from becoming a reality in the foreseeable future.

The particular controllers we used employ "cyclic code" check system. The transmitting station appends a special code word after the last word of the output blocks. The bits in this code depend in a known way on the bits in the data block. The receiving controller regenerates this code as the data is received, and then compares it to the code sent by the transmitting controller. A mismatch flags a status bit or creates an interrupt signal.

The recovery algorithm that we are using is quite simple. Whenever the receiving controller reports a cyclic error, the associated computer *ignores* the data block just received and instead uses the data received in the previous frame, which is still available in the unpacked buffer. The error is also logged, and, if occurring at the central site, is reported to the remote together with its time of occurrence. This technique should be adequate for digital simulations. Its adequacy in a *hybrid* environment needs to be verified.

Our tests of line reliability under real-time conditions are incomplete at this writing. Since line conditions, load, weather and other factors affecting reliability differ considerably from one area to the next, it is doubtful that whatever results we obtain can be assumed to be valid everywhere. However, transmission error statistics collected during RJE activities at the Boeing Seattle facility indicate that the error rate is less than 1 in 10,000 transmitted blocks (.01%) between the hours of 2 PM and 8 PM, and is practically nil outside those hours. That rate includes some retransmission attempts; and since these retransmissions will not be attempted under real-time conditions, error rates can be expected to be no more than that reported above.

It should be noted, incidentally, that the error checking facilities built into the communications controllers, beside being highly sensitive (detecting essentially all short error bursts), also represent an actual *improvement* over the local AD link, where, usually, *no* error checking of any kind is performed.

HOW THE REMOTE SIMULATION IS ORGANIZED

Timing

Since the primary function of the timing mechanism ("real-time clock" or "interval timer") in a hybrid simulation is to trigger the AD conversion equipment, it would seem to be convenient to place the responsibility for timing the simulation with the remote terminal, particularly when the central computer is of a conventional type. In our experiment, however, the timing function is performed at the central site for two reasons. First, a 6000 peripheral processor easily performs this function. Since one is dedicated to the remote simulation task anyhow, any task added to it is essentially a zero-cost item and, in fact, represents a savings in eliminating the requirement for a hardware timer at the remote site. Secondly, and more significantly, by placing the timing function at the central site we automatically obtained an important debugging capability: namely, the ability to exercise the central-site CPU program under real-time conditions without using the remote terminal or the communication controller. An important benefit of this arrangement was our ability to run extensive throughput tests on the central site computer some two months before the central-site CPU control program and the mathematical model became operational. These tests are discussed later.

Sequence of events

Figure 2 shows the sequence of events occurring during every frame at both ends and their time relationships. In a more conventional computer, the functions performed in the 6000 Peripheral Processing Unit (PPU) will have to be allocated in part to a buffered data channel and in part to the CPU.

At the central site, the PPU detects the beginning of a new frame in its clock* routine and transmits a "Start-of-Frame" interrupt code to the remote. It then

* The clock referred to is the 6000 12-bit real-time clock (a standard feature) counting at 1 microsecond intervals. Our PPU maintains a 24-bit "software" clock by referring to the hardware clock periodically.

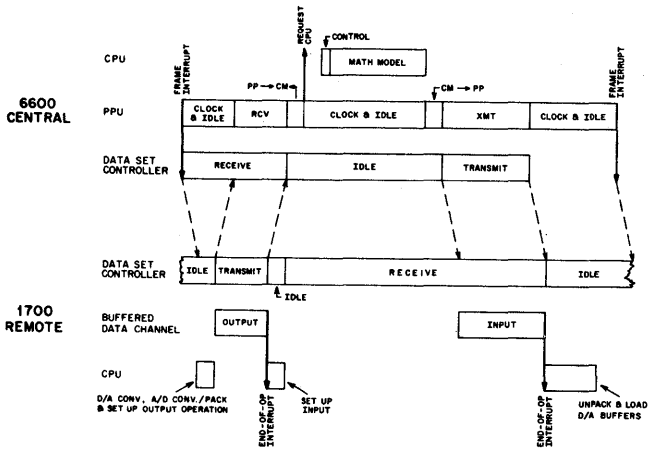


Figure 2—Sequence of events in a frame

This diagram shows the scheme of synchronized communication employed. A Start-of-Frame interrupt from the 6600 triggers a sequence of data exchange in a prearranged order known to both computers. When not required by the real-time simulation, the central and remote CPU's are available for batch or other work.

Not shown is the option of transmitting to the remote previous (rather than current) frame data, to acheive a high degree of I/O—CPU overlap in the central computer. In this option, the phase shift error is 2T, where T is the frame time. Standard extrapolation techniques may be used to compensate for this error.

switches the communications controller to the RECEIVE mode in preparation for the expected AD data from the remote. When this data begins to arrive, the PPU accepts it into a temporary buffer in its own memory. After the correct number of words have been received, the PPU transfers them to Central Memory (CM) and requests the CPU from the monitor. When the CPU program is finished, it signals the PPU, and the PPU transfers the block of output (DA) data just generated by the CPU program to its own memory.** The PPU then switches the controller to TRANSMIT, transmits the output block, and returns to watch the clock for the beginning of the next frame.

The computer at the remote end, with its communications controller at IDLE, waits for the Start-of-Frame interrupt code from the central computer. The receipt of this code interrupts normal processing on the remote computer and transfers control to an interrupt routine. This routine pulses the CONVERT command for the DAC's (digital-to-analog converters), and switches the SH (sample/hold) circuits to HOLD. It then inputs the AD values (a buffered channel operation can be used) and packs the data in a format which will result in the most efficient transmission, i.e., the least number of unused bits. This is particularly important when using the 40.8 K bits/sec. facility, since at this relatively low speed, data transmission can occupy the major portion of the frame time. The routine then releases the SH's, switches its controller to TRANSMIT, initiates a buffered data channel output operation, and returns control to the operating system.

The end-of-operation interrupt from this channel calls another routine. This routine switches the controller to RECEIVE, in anticipation of DA data to be sent by the central computer, and initiates a buffered input operation, after which control is again returned to the system.

The data channel commences input only when the Begin Data code arrives. This time, the end-of-operation interrupt, which occurs after the last data word is received, schedules a routine which unpacks the DA data just received and loads it into the first ranks (buffer registers) of the DAC's, in preparation for the next Start-of-Frame interrupt code, which will cause actual DA conversion. The controller reverts to IDLE automatically when the reception is complete. The remote now waits for the next Start-of-Frame interrupt code.

Synchronization and central program organization

The remote communication equipment imposes certain restrictions on the method of communication between the two computers. It is important to recognize these restrictions, since they are essentially independent of the type of computers involved.

1. No control, status, or parity information is carried *within* the data words or block. This is done in the interest of speed.
2. Control information can be exchanged only when data is *not*.
3. Typically, only one control (interrupt) code is available to obtain the other computer's attention.
4. The controller does not know the size of the data blocks.

The practical implications of these restrictions are that the computer cannot *function* the remote equipment, that the remote equipment cannot *interrupt* the central computer with a priority structure and that both computers must be aware of the size of the data blocks exchanged.

** Optionally, the PPU can be made to transmit data from the *previous* frame, overlapping the transmission with CPU execution for the current frame. This permits increasing the volume of data exchanged (or, alternately, increasing the permissible CPU execution time) every frame, at the cost of increasing the phase shift error to twice the frame time.

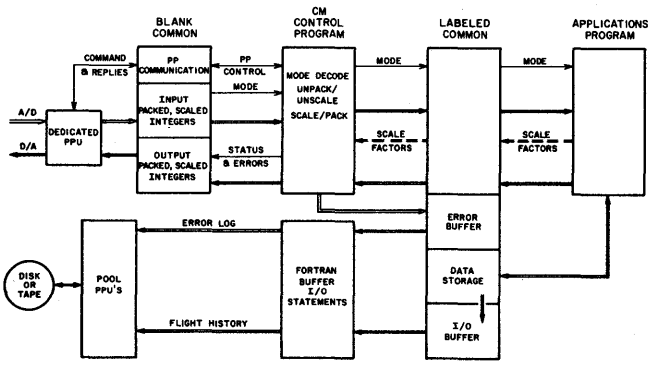


Figure 3—CM program organization

The user's (applications) program, which is formally a subroutine of the control program, communicates with the latter through labeled COMMON blocks. The control program handles system communication tasks, controls the PPU remote communications program, and provides scaling and packing services. Under control program direction, user's flight history data, as well as error and timing statistics, are directed to disk or tape files. The remote computer is transparent from the user's point of view and requires no programming or special considerations.

Two *logical** solutions to this impasse are possible. The one typically employed in RJE support-software, uses the interrupt code as an "attention getter", followed by the exchange of a block of data of *known length*. This block contains either *instructions* on what is to follow, or *requests* for desired action. In order to prevent confusion, one computer is typically restricted to issuing instructions only, whereas the other issues requests only.

The second solution, which—as is evident from the preceding discussion dealing with the sequence of events—is the one that we use, is to periodically exchange data in known block sizes, and have the control or status information coded into known places in the data block. This technique is particularly suitable for real-time simulations, which are characterized primarily by the periodic exchange of blocks of data between the analog and digital domains.

In our case, the first word of both the input and output blocks is reserved for functions and status. Typically, the remote computer codes into its outgoing block mode control and other requests introduced by the remote user through his typewriter keyboard. The central computer uses the control word for mode acknowledge (to signify the acceptance of the remote user's commands) and for error information, alerting him to errors occurring either in his central program or in the received data blocks.

Perhaps the most important outcome of this method of synchronization is the organization of the CM (Central Memory) program. Figure 3 shows the overall data flow and program organization in CM. The requests contained in the mode word received at the central computer often require further dialog with the PPU (somewhat like system action requests in more conventional computers); at other times, these requests need only be interpreted by the applications program (for example, a mode change from OPERATE to IC); and sometimes both system action and applications program cooperation is required. To relieve the user of the need to communicate with the PPU or the system, a central memory control program was introduced. This central control program (coded entirely in FORTRAN) acts somewhat like a monitor or a supervisor as far as the applications program is concerned, but appears to be just another user's job to the system's (real) monitor. Rather than have the user communicate with the system and PPU through FORTRAN callable subroutines, as is normally the case in conventional real-time simulations, the user's program is, formally, a *subroutine* of the control program.* The control program, which is called by the PPU at the start of every frame, examines the mode word in the input block just received. The control program initiates system or PPU actions, if required, before transferring control, with a minimum of fanfare, to the user.

The interface between the user's program and the control program is in labeled COMMON, to which the user supplies arrays of scale factors and unscaled, floating output data, and from which he retrieves input data ready for unscaled floating computation, as well as mode information.

The resulting modular program organization not only frees the user from the drudgery of handling complicated, machine-dependent control functions so that he is able to concentrate on developing his mathematical model program, but also permits replacing one application program by another with a minimum of pain.

The periodic data exchange, by keeping the communication link alive at all times, permits the IC mode to become an important debugging tool, since information flow is dynamic but the central applications

* The hardware solutions: use a communication link that will sustain the same data rate as local equipment, and design data sets and controllers to decode control information; or, dedicate a separate communication link for control and status. Either solution is very expensive.

* Obviously, it is just as easy to require the user to execute a call to the control program as his first executable statement. Aside from psychological considerations, there seems to be no real advantage to doing so.

program is static. Furthermore, a *static test* mode is easily implemented without requiring special programs to be loaded in either at the remote or the central site. In this mode, blocks of bit patterns are circulated through the entire system and searched for errors when they arrive back at the originating computer. The integrity of the communication path, as well as the reliability of the AD equipment, can be easily checked in this mode.

Simulation control at the remote site

The remote user is presently provided with an elementary level of control and monitoring of his job. Through the keyboard of his remote computer, he is capable of entering commands of essentially two types:

1. Simulation Mode Control
2. On-line debugging

Mode control commands include the standard OPERATE, HOLD, and IC which are meaningful primarily to the applications program. In addition, two special modes are available: STATIC TEST, which, as mentioned previously, is a hardware test mode intended to check the integrity of the communications path; and STANDBY.

The user enters the STANDBY mode whenever he anticipates a relatively prolonged period of inactivity (say 15 minutes or more) because of the need to attend to the simulation gear, or simply to analyze the results of the last run and prepare for the next one.

The function of the STANDBY mode is to permit the central computer to minimize the idle resources dedicated to the simulation. Upon receipt of this command the CPU is released. The PPU program copies the contents of the core space occupied by the applications and control programs to a file on the disk. It then releases the core space. Then, the PPU program goes into a "periodic recall" state, in which the PPU is released back to the system and is only claimed again for a very short period at regular intervals (nominally every 10 seconds). When the remote user is ready to resume operation, he reinitializes the remote computer; the PPU program detects this and resumes continuous control of the PPU. It then reclaims the necessary memory space and copies to it the program image previously saved on the disk. If, in the meantime, memory has been occupied by other jobs, operator attention is requested via a flashing message on the operator's CRT station. Automatic rollout of batch jobs under these circumstances can be implemented.

For on-line debugging, the remote user can select any variable from his central memory program and direct it to appear on any output (DA) channel by typing in an appropriate command. This has been achieved in a fairly unsophisticated way. The user is required to arrange *all* his variables of interest in COMMON blocks, and order the blocks so that each variable can be referred to by an ordinal relative to the beginning of the first COMMON block. A pointer array is maintained (also in COMMON) with an entry for each output (DAC) channel available. The contents of these pointer cells are the ordinals of the variables which are to appear on the corresponding channels. After completing its computation for the current frame, the applications program fills its output buffer as instructed by the pointer array. The remote user simply manipulates the pointer array contents when he wants to change channel assignment.

A similar technique may be used to *insert* values into any desired variable. The remote station would probably transmit to the central computer the characters as entered by the user. The central computer will perform the conversion to binary quantitites so that the remote computer, which, in general, can be expected to have a much smaller word-length, is not required to carry triple or quadruple-precision quantitites. Digital values to be displayed on the remote computer's typewriter will be treated similarly.

In general it is clear that a much more sophisticated debugging facility is highly desirable, including a symbolic insert/delete facility, as well as the ability to manipulate source files (or any other files for that matter). These capabilities are currently fully developed for local simulations, via an interactive CRT-keyboard user's station. A number of possibilities of bringing these capabilities to the remote site exist.

The remote user is also permitted to manipulate on-line, the *frame time* and the number of AD/DA channels.

Real-time I/O

In almost every real-time simulation, it is desirable to input and output other than analog data during the simulation. This is done, for example, to provide some means of printing flight history data while the simulation is in progress, or, at least, shortly thereafter. Other uses for this capability are the accumulation of error history files and, on the input side, the collection of telemetry data, the reading of functional tables for function generation tasks, and so on.

Standard FORTRAN coded or binary I/O statements cannot be used, since the compiler, taking into account the multiprogramming environment in which the object code will operate, codes these so that control

is returned to the operating system until the requested I/O is completed. Fortunately, the 6000 FORTRAN includes as a standard feature buffer I/O statements which allow the user to continue his processing after initiating the operation. The user must, of course, be careful to check the status of the previous buffer operation before attempting a new one to the same device. In general, the user is expected to use circular (chained tail-to-head) buffers, so that when one segment is in the I/O process, the other is being manipulated by the program. By increasing the size of the buffer, the user can relax his I/O timing constraints, often to the point where he can share the system's device (typically a disk), rather than requiring a dedicated unit (tape or disk pack).

In our experiment (Figure 3) we used this technique for accumulating flight history, error logs, and timing statistics on the system's disk. The files generated in this way can be transmitted to the remote for printout after the simulation.

In the future it may be very desirable to obtain *on-line* printout at the *remote* site. This can be achieved by adding a few words at a time to the output block. However, the number of words per frame that can be transmitted is clearly limited by the speed of the communications facility.

Remote terminal organization

It is worth noting that the system configuration shown in Figure 1 was designed to support remote job entry (RJE) facility under control of a combined 6000/1700 software system termed Export/Import. This facility, which is still available, although not simultaneously with real-time operation, must be regarded as mandatory. It is hard to visualize a serious effort to develop a real-time application remotely, unless the capability for volume source statement insertion and volume printing is present.

In later phases of the development, such as the debugging phase, the ability to quickly insert or delete a few statements at a time, into or from a source file maintained at the central site, becomes more important. Here an interactive CRT station could become a quite useful adjunct to the remote terminal.

In general it is safe to say that a general purpose computer is almost mandatory at the remote site (as opposed to special purpose hardware), to support the many varied activities ancillary to the real-time simulation. These include, among others, servicing of the user's station (typewriter or CRT/keyboard), and support of a card reader and a line printer for the RJE facility. Admittedly, the remote computer need not be as powerful as the one we used. However, if one of the many available "mini computers" is selected, the question of software development for both the simulation tasks and the related activities should be carefully weighed against the possible saving in mainframe cost.

EFFECT ON THE CENTRAL SITE

The installation is administered primarily as a very-high-volume, very-high-throughput facility.* Some 1200 batch jobs, submitted locally and remotely, are processed in a typical day. For this reason it was considered important to establish the effects of adding the remote real-time simulation load to this system. The primary objective of the throughput tests was to obtain quantitative data relating to these effects.

Test plan

The installation maintains two types of standard test devices that enable it to measure the effects of proposed changes—such as modifications to the operating system or installation procedures—on the current load. One such device is the Job Mix Sample. It is a set of some 70 actual jobs whose processing requirements are based on the known characteristics of the current system load. The first shift part of the job mix consists primarily of small, fast jobs, while the 24 hour mix also contains several large jobs with relatively long execution times. The Job Mix Sample is updated about every six months to reflect the current job mix characteristics. In effect, the Job Mix Sample compresses a whole day's operation to a several hours' run, and enables the installation to conduct controlled experiments.

The second device is an event-oriented (discrete) Simulator. The Simulator accepts (a) a description of the important parameters of the computer (e.g., core, number and speed of tapes and unit record equipment, etc.); (b) a definition of the queue-selection and priority determination algorithms; and (c) statistical distributions of job requirements, categorized by user groups, remote or local terminals, tape or non-tape jobs, etc. Job requirement distributions include arrival time and required resources. The Simulator samples these distributions randomly to create a job mix, and monitors job throughput, resource utilization and turnaround times. The Simulator processes a 3 days' load in 30 CPU seconds. Its accuracy is reported to be $\pm 5\%$ on throughput and $\pm 15\%$ on turnaround time.

* The interactive terminals, operating under SHARER, a special sub-system of the operating system, are few and are served only during certain hours of the day.

TABLE II—6600 Throughput Test Conditions

Condition No.	RTS Requirements		Export/Import	Remarks
	Core	CPU	Running	
1	—	—	Yes	Current Production System
2	40 $K_{(8)}$	—	No	Real time job not executing
3	50 $K_{(8)}$	—	No	Real time job not executing
4	40 $K_{(8)}$	25%	No	
5	40 $K_{(8)}$	40%	No	
6	40 $K_{(8)}$	40%	Yes	
7	20 $K_{(8)}$	25%	No	

We used the Job Mix Sample to determine the effect that the real time application had on the batch processing workload of the central facility. We first obtained actual throughput time under normal operating conditions and again under various conditions with the real time job active. Data points were obtained for the first shift portion of the Job Mix Sample, as well as for the entire 24-hour period Sample. The conditions for which these throughput tests were run are shown in Table II.

Test results

We have defined throughput degradation as follows:

$$\text{thruput degradation (percent)} = \frac{t_{(N)} - t_{(ref)}}{t_{(ref)}} \times 100$$

where: $t_{(N)}$ is the time required to run the Nth job mix case and $t_{(ref)}$ is the time required to run the job mix under the current production system. Thus a 100% thruput degradation refers to the situation where, because of the real time job requirements, the job mix takes twice as long to run as it does under the current operation system. Table III summarizes the results of the Job Mix Sample throughput tests.

Data collected from the 6600 simulator program are plotted in Figure 4. This graph depicts the throughput degradation that can be expected as a function of, and because of, the core required by a real time job. The curve passes through zero degradation at a real time job core requirement of $10K_{(8)}$ because this is the amount of core required to support the RJE in the current production system, and RTS and Export/Import do not run concurrently, as explained earlier. Several data points obtained from Job Mix Sample test cases are also shown in Figure 4. These include the results from case 2 and 3 as well as extrapolated data points based on the difference between cases 4 and 7. It will be noted that the correlation between the simulator results and the results of 1st shift thruput tests is quite good while the 24 hour job mix thruput results lie considerably further from the simulator curve. The statistical representation of the 24 hour job mix is probably not nearly as good as that of the 1st shift mix. Thus the discrepancies in Figure 4 are caused by too small a sample of jobs in the 24 hour mix.

Figure 5 shows the throughput degradation resulting from the real time job CPU requirements. The curves are plotted by subtracting the degradation due to RTS core requirements from the total degradation observed during the Job Mix Sample tests. This assumes that total system degradation can be obtained by merely adding the effect shown in Figure 4 to that shown in Figure 5. Two sets of curves are shown in Figure 5. One is drawn under the assumption that the simulator results (Figure 4) are applicable while the other is drawn assuming that the Job Mix Sample test data points apply.

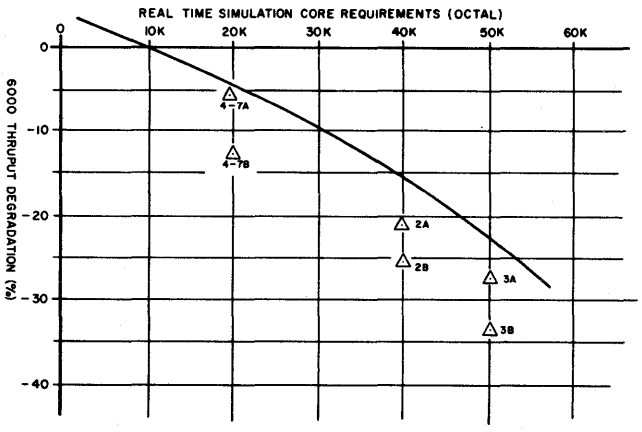


Figure 4—Throughput decrease vs. RTS core requirements

Since RTS replaces RJE packages requiring 10 K_8, the curve passes through zero decrease at that point.

TABLE III—Results of 6600 Job Mix Sample Throughput Tests

Test Condition No.	Run Time (minutes)		CPU Utilization	Thruput Degradation	
	1st Shift Job mix (Case A)	24 hour Job mix (Case B)	24 hour Job mix (Case B)	1st Shift Job mix (Case A)	24 hour Job mix (Case B)
1	33.55	75.28	60.4%	—	—
2	43.00	94.50	50.1%	21.0%	25.6%
3	45.43	100.75	47.6%	27.8%	33.7%
4	47.50	113.00	73.1%	33.6%	50.1%
5	55.30	131.40	83.6%	54.6%	74.6%
6	61.80	142.22	78.5%	74.0%	88.9%
7	41.65	103.30	N/A	17.4%	37.2%

Figure 5 is plotted for a particular configuration of the 6600 system; i.e., no Export/Import operation, RTS assigned a dedicated PP and RTS assigned $40K_{(8)}$ memory locations (which leaves $300K_{(8)}$ for batch work). The CPU time is also assumed to be required on a 20 millisecond interrupt basis. However, included are the results obtained from case 7. Since the real time job was assigned only $20K_{(8)}$ for case 7, and there is good correlation between the results of case 7 and the curves of Figure 5, Figure 5 is probably applicable over a wide range of real time job core requirements.

Figure 6 shows the average total 6600 CPU usage (in percent) by both the batch jobs and the real time simulation while the Job Mix Sample tests were running. Ideally, if every job used the CPU only when no other job needed it (i.e., during the other jobs' I/O wait periods), then 100% CPU utilization could be attained. As this figure shows, while there has been a dramatic improvement in CPU utilization due to the introduction of RTS, there are still occasions when, because of I/O or memory conflicts, no job is ready to use the CPU.

One unexpected result of the throughput tests was the amount of CPU overhead we experienced in getting the real time simulation on and off the CPU. In theory the "on" delay is not accountable as real time job CPU time and the "off" delay, which is accountable, averages about 300 microseconds each time the real time job releases the CPU. The accountable CPU overhead measured during the throughput tests averaged 900 microseconds per frame. We have not been able to identify the source of this overhead.

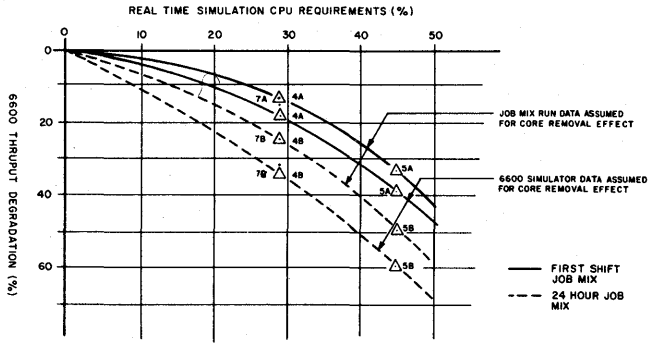


Figure 5—Throughput decrease vs. RTS CPU requirements
Solid lines refer to first shift conditions, dashed lines to the 24-hour case.

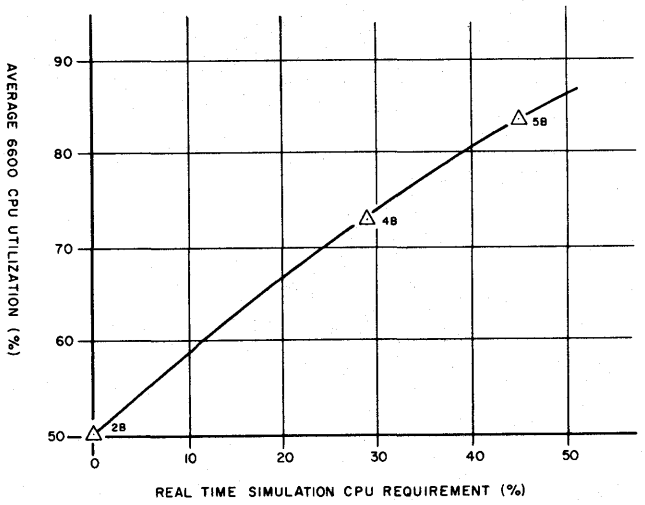


Figure 6—Average CPU utilization vs. RTS CPU requirement

Although percent utilization has increased considerably, it falls short of the theoretically possible 100%. That goal can be attained only if the queue managing algorithms are continuously adjusted to eliminate the condition where, because of I/O waits or memory conflicts, no job is ready to use the CPU.

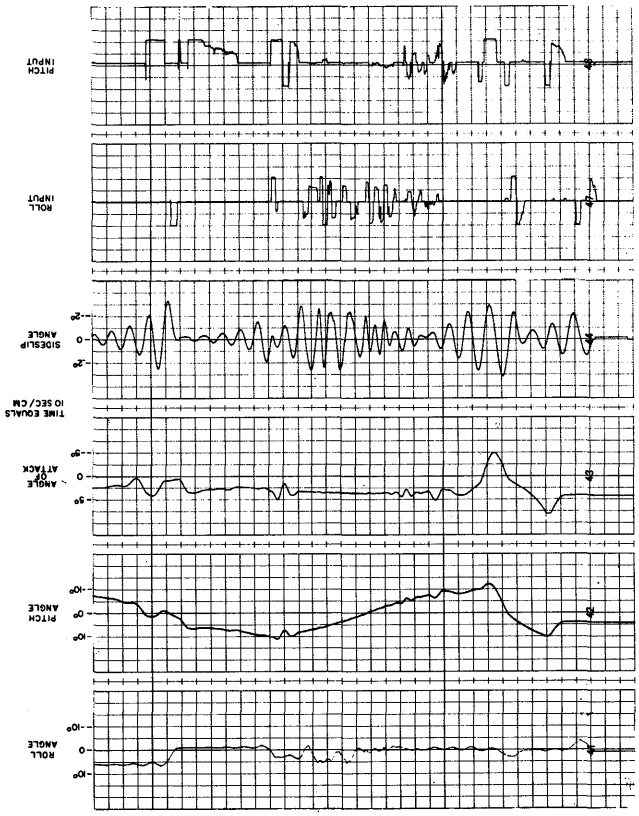


Figure 7—Typical strip chart recordings from the simulation

Roll and pitch commands are generated by a "joystick" and transmitted to the central 6600. Other traces represent results of the 6600 computation transmitted back to the remote site.

Using the results

Knowing the extent to which a real time application will affect the batch throughput on a central facility is one thing, but determining whether the operation could tolerate the degradation is quite another. A host of variables such as "spare" capacity on the central processor, the percentage of the time during which the real time job is inactive ("STANDBY"), relative economies and flow time associated with alternate approaches, as well as importance of the real time application, make sweeping generalizations rather difficult. It is possible, however, to identify several situations where the remote application concept would be an attractive solution to a real time computing requirement.

A central facility operating at less than 85% or 90% capacity (critical resource capacity) is, of course, an excellent candidate for implementation of the remote simulation capability. Once a production facility is operating above that level, the relative merits of batch vs. real time work come into play; but several conditions could result in the real time job taking precedence. Some of these conditions might be:

1. Limited duration of the real time application
   a. to support a "crash" effort (e.g., short-deadline proposal)
   b. to conduct a one-shot series of tests
   c. to allow time for purchase of additional computing equipment.
2. Availability of "outside" facilities to run batch backlog created by the real time application.
3. Restricting the real time application to certain hours of the day.

Special test facilities or hybrid simulation labs that are currently in the planning stages can quite easily be structured to take advantage of the remote simulation concept, since remote terminal processors that can double as special purpose stand-alone digital computers are available. In a lab such as this, much of the real time application work would be accomplished using the remote terminal processor only, but the speed and power of a large central processor would be available for those applications that required additional computing capacity.

CONCLUSION

A successful demonstration of remote real-time simulation was carried out in January, 1970. The demonstration consisted of "flying" the airplane simulation discussed in this report from the developmental center (location of the 1700) in Seattle, Washington. The math model was implemented on a 6600 in the Boeing Renton facility and communication between the 1700 and 6600 was via 48 KHz lines. Each mode of operation discussed earlier was shown to be operational. Figure 7 is a typical set of strip chart recorder traces obtained during the demonstration. Traces of the pitch and roll commands as well as computed roll, pitch, sideslip angle and angle of attack shown. This technique is presently being considered as the primary simulation tool for possible forthcoming work such as the B–1 effort.

The remote simulation concept, therefore, appears to offer important new possibilities in the design of new simulation facilities and in the manner of utilization of existing ones.

ACKNOWLEDGMENTS

Messrs. A. Ayres and Dennis Robertson, of the Boeing Company, coded the mathematical model for the CDC 6600 central computer.

Mr. Leo J. Sullivan, assisted by Mrs. Roberta Toomer, both of Control Data Corporation, coded the software for the CDC 1700 remote terminal and the CDC 1500 analog-digital link. Mr. Sullivan and one of the authors (O. Serlin) performed the actual checkout of the system.

Mr. Robert Betz, of Boeing, performed the throughput tests on the 6600 and assisted in analyzing the results.

Of the authors, O. Serlin is responsible for the conceptual design of the software, as well as the coding of the 6600 central control program, operating system modifications, and PPU program. R. C. Gerard planned and analyzed the throughput testing experiments, conducted the demonstration, and coordinated the Boeing Company's efforts on this project.

Many other persons contributed to the project in other than the technical areas. Among these, Messrs. John Madden, of Boeing, and Tim W. Towey, of Control Data, deserve special recognition.

REFERENCES

1 M S FINEBERG O SERLIN
Multiprogramming for hybrid computation
Proc AFIPS FJCC 1967

2 J E THORNTON
Parallel operation in the Control Data 6600
Proc AFIPS FJCC 1964 Volume II

3 Control Data® Publication No. 60152900
31700 Computer System Manual

4 J R BAIRD
An optical data link for remote computer terminals
Datamation January 1970

Real time space vehicle and ground support systems software simulator for launch programs checkout

by H. TRAUBOTH

Marshall Space Flight Center
Huntsville, Alabama

and

C. O. RIGBY and P. A. BROWN

Computer Sciences Corporation
Huntsville, Alabama

INTRODUCTION

The launch of a Saturn V vehicle is preceded by a complicated chain of checkout operations, which involve a large system of checkout and launch equipment in the Launch Control Center and in the Mobile Launch Facility at the Kennedy Space Center to assure the integrity of the flight systems. This checkout and launch system consists of manual checkout panels, ground support equipment (GSE), telemetry stations, data links, and two RCA–110A launch control computers. Commands initiated in the Launch Control Center are transferred by these computers to the launch vehicle under checkout. The computer sends out stimuli and receives responses which are evaluated based on predicted values stored in the computer memory. Sending out stimuli and monitoring the responses is done in a controlled sequence by test programs residing in the launch computers. These test programs must be thoroughly checked out before they are allowed to run at the launch facility. The rigid testing of the launch computer programs is done at simulation facilities which imitate as closely as possible the environment of the launch computers, i.e., the functions of the vehicle, of the GSE, and of the checkout system. Most of the checkout is done with hardware simulators such as the "Saturn V–Breadboard" which uses partly actual flight hardware and simulates certain mechanical and hydraulic equipment by electrical circuits.

In order to aid the checkout engineer in the design and evaluation of test programs, two major software simulators have been developed by MSFC. These software simulators simulate the on-off functions of discrete networks by evaluating large sets of Boolean equations including discrete time-delays for pickup and dropout of relays, valves, etc. They evaluate the equations in non-real time and are driven by pre-determined sequences of states of switches and stimuli as generated by test programs.

More than three years ago, a joint effort between the Astrionics Laboratory and Computation Laboratory began to define a simulation system in which a digital computer would simulate in real-time the vehicle and GSE functions in response to stimuli sent from the two launch computers. The objective of this project was to find a new way to perform major functions of the Saturn V–Breadboard with a more flexible digital computer, so that RCA–110A launch computer programs could be checked out, test programs could be evaluated, and the effect of malfunctions could be investigated without having to use and possibly damage expensive hardware. The primary design objectives were to insure that:

1. The simulator would act in such a way that the test programs of the two launch computers would think they were working with the actual vehicle and GSE in real-time.
2. The prime portion of the simulator, the software, should be structured in such a way that no reprogramming would be necessary when a configuration other than Saturn V had to be simulated, as long as the hardware components could be described by the same nomenclature for the data base.

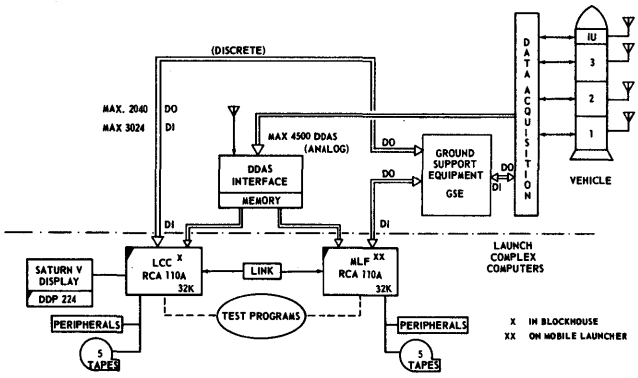


Figure 1—Saturn V-launch computer complex configuration

3. To provide the engineer with the capability of communicating directly with the computer when using the simulator.

It was determined that the feasibility of this approach could best be demonstrated by using the Saturn V configuration as a test bed.

The emphasis of this paper is on describing the software of the simulator. The operation of the simulator facility and the form of the mathematical models which are input into the computer are described in detail in Reference 5. However, to understand the structure and problem areas of the software, it is necessary to also understand the configuration of the hardware.

SCOPE OF SIMULATION

A simplified diagram of the Saturn V–Launch Computer Complex configuration is shown in Figure 1. The launch computers in the Launch Control Center (LCC) and in the Mobile Launch Facility (MLF) send out discrete signals (up to 2040 "Discrete Out or DO") to the Vehicle through the GSE. The sequence and addresses of these signals is determined by the test programs. The vehicle then sends discrete and analog responses (measurements) back to the computers. Most of the discrete measurements (up to 3024 "Discrete In or DI," i.e., open/closed relay contacts, valves, switches, or gates) are fed through the GSE, while all the analog measurements and a few digital measurements are transmitted through the digital data acquisition system (DDAS) or telemetry system into the DDAS Computer Interface. The DDAS is the whole collection of equipment which lies between the sensors and the DDAS Computer Interface, i.e., per vehicle stage a transmitter; a line driver and receiver; and a digital receiver station. The transmitter consists of a scanner, digital and analog multiplexer, A/D converters, generator of identification codes, and modulators; the line driver and receiver contain amplifiers and demodulators. The digital receiver station converts the demultiplexed measurement information into synchronized data words and address words and sends them to the Computer Interface. The Computer Interface is mainly a digital memory that can store up to 8192 words, and a special controller which stores one measurement word every 278 μsec in proper sequence and which allows the launch computers to retrieve stored data at random under several modes. The data are stored in the Interface according to their identification number containing the stage, channel, frame, multiplexer, and master frame numbers. The controller insures that the data request from the RCA–110A computer is properly decoded to find the requested measurement. For a measurement which has to be scanned at a higher rate, the scanner, moving with constant speed, accesses the sensor several times, and therefore, stores it at several places. The RCA–110A computer can access the data in the Interface in several modes, e.g., the request can be synchronized with the incoming data, or locked at a specific measurement. Up to 4500 DDAS measurements can be handled by the Interface.

The launch computers themselves are connected through a data link for exchange of information. The test conductor controls the launch checkout through the Saturn V–display which is driven by a smaller DDP 224 computer. The coordination of the many test programs, display programs, and control programs for the peripheral equipment (printer, card reader, etc.) is done by the RCA–110A Operating System.

The simulator performs the functions of the equipment shown in the upper portion of Figure 1. The

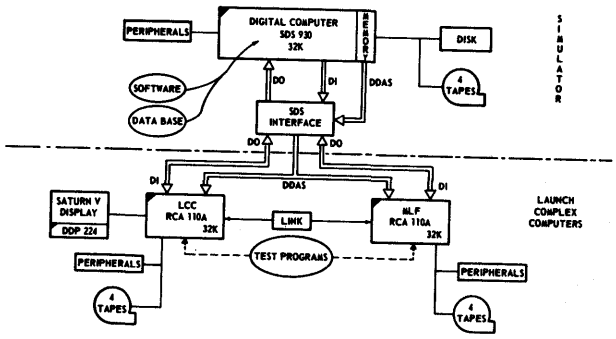


Figure 2—Real-time simulator systems configuration

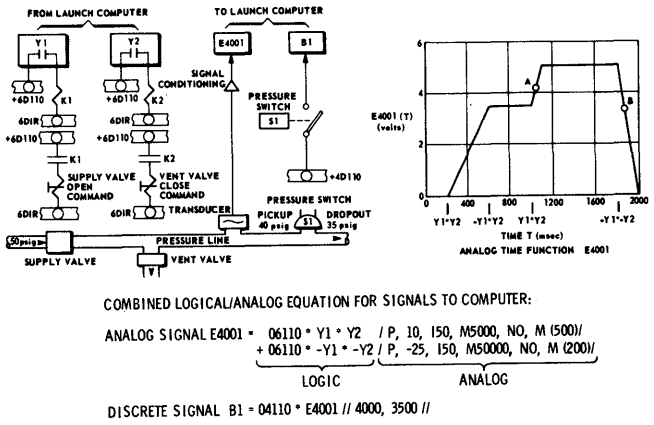


Figure 3—Example of a typical discrete/analog circuit

hardware portion of the simulator comprises the SDS–930 digital computer with 32K words of core memory and its peripheral equipment and bulk memory devices, and a special purpose interface (SDS Interface) which is in size similar to a small computer (Figure 2). This interface performs the functions of the DDAS Computer Interface but does not contain a memory since a portion of the SDS–930 memory is dedicated to these functions. The SDS Interface contains counters, special registers, and controllers which enable the two launch computers to communicate with the SDS–930 computer in the same modes as in their actual launch complex environment.

Data base

The functions of the vehicle and its ground support equipment as seen by the test programs can be described by large sets of logical equations and by analog time functions which are described by polynomials or tables. The logical or Boolean equations are time-dependent in the sense that they consider pick-up and drop-out time as a time-delay. The logical equations consist of AND and OR terms (* and +) and negations of a single variable (−). Special relays such as lock-out, lock-up, and latching relays can be expressed by equivalent circuits of regular relays. Figure 3 shows a simplified example of a typical discrete/analog circuit. For more detailed information see References 5 and 7.

There are basically two types of equations possible:

Logical equation

$$E^{(i)} = Y_{11}{}^{(i)} * \cdots * Y_{K11}{}^{(i)} + Y_{12}{}^{(i)} * Y_{22}{}^{(i)} * \cdots * Y_{K22}{}^{(i)} + \cdots + Y_{1a}{}^{(i)} * Y_{2a}{}^{(i)} * \cdots * Y_{Kaa}{}^{(i)}$$

where

$$Y_{pq}{}^{(i)} = (-)Z_{pq}{}^{(i)}(P_{pq}{}^{(i)}, D_{pq}{}^{(i)})$$

or

$$Y_{pq}{}^{(i)} = (-)Z_{pq}{}^{(i)} \,||\, P_{pq}{}^{(i)}, D_{pq}{}^{(i)} \,||$$

$P_{pq}{}^{(i)}$ = Pick-up time (amplitude) of element $Z_{pq}{}^{(i)}$

$D_{pq}{}^{(i)}$ = Drop-out time (amplitude) of element $Z_{pq}{}^{(i)}$

and i, p, q, and a are unlimited index integers 1, 2, 3, . . ., i.e., the number of equations, OR-terms, and AND-terms is not limited. Pick-up time for a relay means the time between activation of the coil and the closure of an associated contact. Drop-out time is the time between deactivation of the coil and opening of an associated contact. Generally, pick-up time is the time-delay between cause and effect, and drop-out time the time-delay of the reverse action. Or mathematically, if t_1 is the time instant of activation (deactivation) the discrete variable $Z_{pq}{}^{(i)}$ is

$Z_{pq}{}^{(i)} = 0$ for time $t < t_1 + P_{pq}{}^{(i)}(t \geq t_1 + D_{pq}{}^{(i)})$

$Z_{pq}{}^{(i)} = 1$ for time $t \geq t_1 + P_{pq}{}^{(i)}(t > t_1 + D_{pq}{}^{(i)})$

For a relay which has a pick-up/drop-out time of less than 10 msec the time-delay is ignored because the delay does not have an effect on slower mechanical devices. Thus, relay races between fast relays cannot be simulated, and it is not intended to detect them because the test programs do not check for them. For most relays, the bracket term can be deleted.

The value of a logical variable may depend on the amplitude of an analog value, e.g., the pressure in a line, instead of a time delay. Then we write the terminators || instead of parentheses ( ).

The discrete variables can have different physical meaning. We distinguish between a "DO" (Discrete Output signal from RCA computer), "DI" (Discrete Input signal to the RCA computer), digital DDAS, manual switch, power bus, and an "IV". An "IV" is an internal variable which is needed for internal computation when a circuit stores a signal.

Combined logical/analog equation

A logical equation can be combined with an analog function. However, each analog function can be associated with only one OR-term of the logical equation. An analog function can be described in eight different

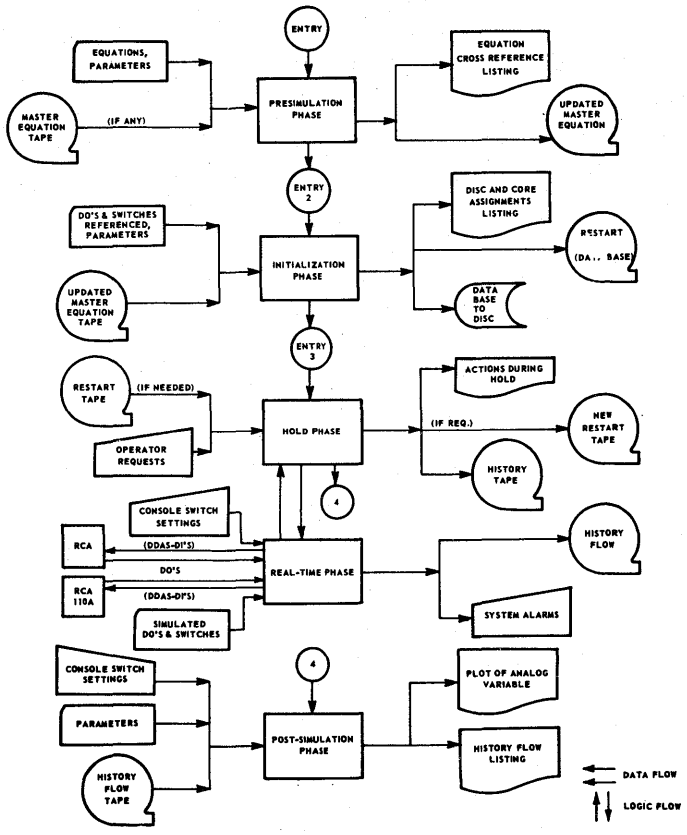


Figure 4—General flow of simulation processor

formats such as a polynomial, table, cyclic function, etc.[5] In any case, the analog function is described to the computer by a one-letter code F designating the type of format and a variant set of parameters, all of which are enclosed with the terminators $|\cdots|$. These parameters also contain the sampling rate for that particular analog variable, the maximum and minimum amplitude limits, the maximum time, and the period of the time function. The general form of a combined logical, analog equation is:

$A^{(i)}$ = logical OR-term 1/F, analog function parameters 1 |
+ logical OR-term 2/F, analog function parameters 2 |.

where $A^{(i)}$ is an analog value.

The interpretation of this equation is as follows: Assuming that only one OR-term generates a "1" at one time (exclusive OR-terms) then the analog function of that particular OR-term is evaluated at the specified sampling rate.

The data base for the total Saturn V including the GSE amounts to about 17,000 equations of various length. (See Table I).

If systems other than Saturn V are to be simulated, only another data base has to be established; no reprogramming is necessary as long as the functions of the physical system can be described by the same types of equations. The data base is initially set up via the card reader; modifications to it and control commands for the simulation are input via teletypewriter.

SIMULATOR SOFTWARE

General

The software for the simulator can be divided into three major areas: (1) Interface Support Software, (2) Simulation Processor, and (3) Simulator Diagnostics. The Interface Support Software controls the input into the DDAS-tables of the SDS–930 core memory and the output from it to the RCA–110A computer supported by the hardware of the SDS Interface. It also controls the transmission of the DO's, DI's, the analog values, various counters, and clocks. The Simulation Processor generates the data base in the computer from the card input, evaluates the equations during the simulation run, and controls the selective print-out of the simulation results. The Simulator Diagnostics checks all hardware units of the SDS Interface such as counters, data link control signals, and data transfer registers, and the communication between the SDS–930 and RCA–110A computers. The diagnostics check especially for critical timing.

The design of the software is modular so that modifications can be made relatively easily. The total software excluding the simulator diagnostics consists of approxi-

TABLE I—Magnitude of Equations

		DO	SWITCH	DI	DDAS DI	BUS	IV	TOTAL
INSTRUMENT UNIT	Discrete Variables	247	557	791	186	69	579	2429
	Analog Variables							505
	Discrete & Analog							2934
	Discrete Equations			791	186	69	579	1625
	Analog Equations							505
ALL STAGES	Variables							about 10000
	Max. Computer Capability	4032	1000	6048	3000	3000	3000	about 20000
	Capability for DDAS (Analog)							4000

LENGTH OF EQUATIONS (Rough Estimate)

Smaller than 10 OR-Terms	about 40%
Larger than 100 OR-Terms	about 20%
Between 10 & 100 OR-Terms	about 35%
Up to 1000 OR-Terms	some %

mately 315 subroutines and 26,000 instructions. These figures should give a feel for the magnitude of the software effort.

Interface support software

As stated previously, the simulator must provide all data input values to the launch complex computers as those provided by the launch vehicle, the Ground Support Equipment, manual checkout panels, mechanical and pneumatic systems, etc., at the Saturn V Launch Computer Complex. Additionally the simulator must accept input data from the launch complex computers and provide the necessary stimuli to the launch complex computers. These stimuli, in the form of Discrete Out (DO) signals, Discrete In (DI), and DDAS signals are provided by the interplay between the Interface Support Software and the SDS Interface.

Each discrete signal (DI and DO) is represented in fixed locations of the SDS–930 memory by the presence or absence of a signal bit in the DI- and DO-Status Table thus allowing twenty-four discretes to be represented in each twenty-four-bit computer word.

Software is also provided to support the input/output requirements to Direct Access Communication Channels, Time Multiplexed Channels, etc., for the storage and retrieval of data from mass storage and the recording of data on magnetic tape.

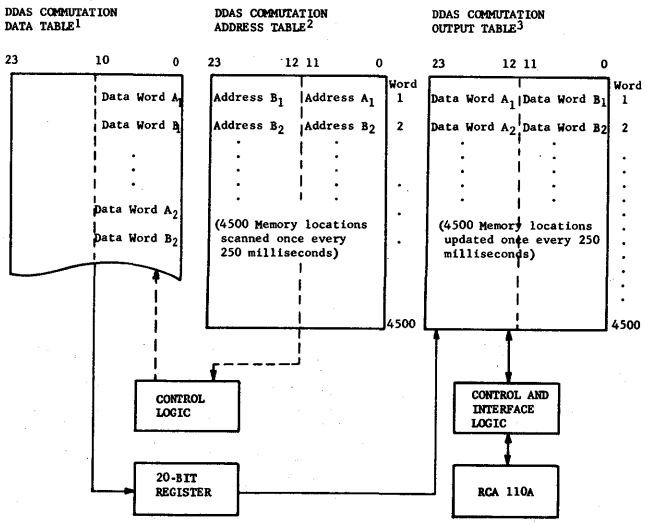
Discrete out/discrete in signals

The Real-Time Phase of the simulation processor receives Discrete Out signals from either of the two RCA–110A computers via the SDS Interface and stores these signals in the DO Status Table. Upon receipt of the Discrete Outs a chain of Boolean and DDAS equations are evaluated by the Real-Time Simulation Program, and the results of the evaluation are placed in either the Discrete In (DI) status table or in the DDAS data table. Both the Discrete In Status Table and the DDAS Data Table are scanned continuously by the SDS Interface, thereby providing current information to the RCA–110A computers upon request.

Digital data acquisition system (DDAS)

The basic function of the DDAS facility at the Saturn V Launch Computer Complex is to periodically sample vehicle parameters and make this real-time data available to the two RCA–110A launch computers. In a simulated environment, the Simulator and its associated Interface hardware must commutate data for use by both RCA–110A checkout computers. This commutated data must be in the same format as the data provided by the Launch Computer Complex. This will allow the Simulator to provide information for the RCA–110A through the commutation processing of the SDS Interface. The DDAS data represent both analog and discrete data. The analog data is represented in ten bits of a twenty-four bit SDS–930 computer word. There are ten discretes represented in each SDS–930 computer word with each discrete being represented by the presence or absence of a single bit.

DDAS simulation requires three DDAS memory tables within the SDS–930 computer for use by the Simulator and the SDS Interface (Figure 5). The DDAS data words which are the results of the evaluation of the combined discrete/analog equations are stored in a block of memory of the SDS–930 computer which is referred to as the DDAS Commutation Data Table. The address of this data word is stored in the DDAS Commutation Address Table according to the sampling rate required for this measurement. As a final step in the commutation process, the data words are stored by the SDS Interface in the appropriate locations in the DDAS Commutation Output Table where they can be accessed by either RCA–110A computer via the SDS Interface.



1. Table is updated by the Simulator Program. Reserved during Initialization Phase.
2. Table is used by interface hardware for control of data transfer from Data Table to Output Table. Setup during Initialization Phase.
3. 24-bit words available to the RCA 110A's from this table. All accesses by the RCA 110A is through the interface hardware. Reserved during Initialization Phase.

Figure 5—DDAS simulator commutation memory tables

Time counters and clocks

Two methods of timing are provided in the Simulation Processor to establish the necessary control for scheduling timed events during the Real-Time Phase of simulation.

A set of 190 *elapsed time counters* provide the capability of establishing arbitrary time delays for scheduling of time-critical equation evaluations or re-evaluations. The counters are *fixed*, consecutive cells in the SDS–930 memory which are incremented by the interface hardware at one millisecond intervals. The number of counters is limited to 190 because incrementation of each counter requires two cycles (3.5 microseconds) of memory access time (maximum of 0.665 ms, leaving 0.335 ms for DDAS commutation and for servicing RCA–110A data requests). The Real-Time Simulation Phase stores the complement of the desired elapsed time in a pre-defined memory location and initiates the hardware incrementation which is then automatic until a cell counts to zero. When a cell has counted to zero, the automatic incrementation halts, the address of the zero cell is recorded by the SDS Interface, and a program interrupt occurs. The Interrupt Service Routine then schedules the associated equation for evaluation by placing it in the highest priority queue. The Interrupt Service Routine also removes this clock from the queue of active clocks and reinitiates the automatic incrementation. The automatic incrementation must be restarted within one millisecond to insure that all clocks are updated accurately.

The system *real-time clock* provides the relative time of events during the Real-Time Phase of the simulation processes. The real-time clock is serviced at one millisecond intervals by a system generated interrupt. At each interrupt, the Real-Time Program increments the real-time clock.

Disc/tape-core communications

The *Direct Access Communication Channel* (DACC) controls the transmission of equations from the rapid access disc (RAD) to the SDS–930 memory. Initiation of the transmission is controlled by the Real-Time Simulation Phase which uses the buffer code/disc address of the equation in conjunction with the Buffer Description Tables (Table II) to compute the number of words to be transmitted and then initiates the transmission which remains under DAAC control until completed. Upon completion, a program interrupt is generated. The Interrupt Service Routine transfers the equation to a specified memory location for later evaluation, and initiates the transmission of the next equation to be input.

TABLE II—Buffer Description Tables

BUFFER CODE ADDRESS TABLE

WORD	DEFINITION
0	ADDRESS OF START OF BUFFER GROUP WITH CODE 0
1	ADDRESS OF START OF BUFFER GROUP WITH CODE 1
2	ADDRESS OF START OF BUFFER GROUP WITH CODE 2
3	.
4	.
.	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE SIZE TABLE

WORD	DEFINITION
0	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 0
1	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 1
2	NUM. OF LOC. ALLOTED TO EACH EQU. REC. IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE NUMBER TABLE

WORD	DEFINITION
0	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 0
1	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 1
2	NUMBER OF BUFFERS ALLOTED IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

BUFFER CODE AREA TABLE

WORD	DEFINITION
0	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 0
1	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 1
2	TOTAL NUMBER OF LOCATIONS IN BUFFER GROUP 2
3	.
4	.
.	.
.	.
.	.
N	. N = 31

During the Real-Time Phase, the Time-Multiplexed Communication Channel is used for generating a complete history of events to magnetic tape.

Simulation processor

The prime objective of the Simulation Processor is to prepare and execute the simulation in such a way that the stimuli of the RCA–110A computers are all received and their response signals (often several responses to one stimuli) generated with the same precise time lag as in reality.[13,14] The huge size of the equation data base (17,000 equations), the limited core size of actually only 19k words (13k of the 32k are used by the DDAS decommutator), the relatively long average access time of the disc memory (17 msec), and the mini-

mum response time of some circuits to be simulated (about 100 ms) constrain the design of the Simulation Processor considerably. Considering the above constraints and in order to minimize the computation time during real-time simulation activity, as many functions as possible are performed in the Pre-Simulator Phase.

The Simulation Processor is divided into five major phases:

- Pre-Simulation Phase
- Hold Phase
- Initialization Phase
- Real-Time Simulation Phase
- Post-Simulation Phase

An overview of the general flow of the simulation processor is given in Figure 4.

Pre-simulation phase

The Pre-Simulation Phase was designed to perform initial processing for all functions that could be predefined and established prior to execution of the Real-Time Phase. This approach was taken in order to simplify the real-time processing functions and to significantly reduce execution time and core memory space.

The Pre-Simulation Phase consists of 85 subroutines and approximately 11,200 instructions.

The Pre-Simulation Phase consists of four sub-phases which are overlaid during execution in SDS-930 memory. A summary of Pre-Simulation Phase capabilities is shown in Table III.

TABLE III—Pre-Simulation Phase Capabilities

- Edits all equations input to the system.
- Builds a cross reference index of equations versus equation terms.
- Establishes tables for time-related equation terms.
- Arranges input equations into the proper equation calling sequence for real-time processing.
- Merges the cross reference values and the time-related values with the equation calling sequence.
- Using card input values, establishes an assignment file for all DDAS values.
- Produces a Master Equation Tape (magnetic tape) consisting of:
  a. Equation cross reference file
  b. Equation file
  c. Ordered DDAS assignment file
- Lists the Master Equation Tape.

Logical equations:

$$Y_1 = Y_3 + DO*Y_1 + DO*Y_2$$

$$Y_2 = DO*Y_1*Y_3 + Y_2*\overline{DO} + Y_2*Y_1$$

$$Y_3 = \overline{Y}_1*\overline{DO}*Y_2 + \overline{Y}_1*DO*\overline{Y}_2 + \overline{Y}_1*Y_3 + Y_3*DO$$

Variable (= Equation) Name	Initial State	No. of Evaluation I	II	III	IV	Initial State	No. of Evaluation I	II	III	IV
Y_1	0	1	1	0	0	1	1	0	0	1
Y_2	0	0	0	0	0	1	1	1	1	1
Y_3	1	1	0	0	0	1	0	0	1	0
		Stable transition (1)				Unstable transition (2)				

Figure 6a. Results of successive evaluation of logical equations with two different initial conditions if DO changes state from "1" to "0".

Y_1	Y_2	Y_3	DO = 0	DO = 1
0	0	0		
0	0	1		
0	1	1		
0	1	0		
1	1	0		
1	1	1		
1	0	1		
1	0	0		
			(1)	(2) (3) (4)

Figure 6b. Transition table for all possible states.

- denotes unstable state
- o denotes stable state

(1) (3) (4) stable transition

(2) unstable transition

Figure 6a—Results of successive evaluation of logical equations with two different initial conditions if DO changes state from "1" to "0"

Figure 6b—Transition table for all possible states

Phase 0 contains the utility and I/O subroutines for the remaining three sub-phases. This phase acts as the Pre-Simulation Phase monitor and remains in memory during execution of phases 1, 2, and 3 to control overlay and input/output operations.

The functions of the vehicle and the ground support equipment are described by Boolean equations which are punched on cards. These cards, containing the data base for the vehicle to be simulated, are input to *Phase 1* where the equations are edited. During this editing process, all equations which contain an error are listed on the line printer with each error flagged. The equations which are error-free are sorted and all duplicate equations are eliminated. The sorted, error-free equations are then used to update the Master Equation Tape (Figure 9) which contains all equations which describe the vehicle configuration to be simulated. During the Master Equation Tape update, sorted

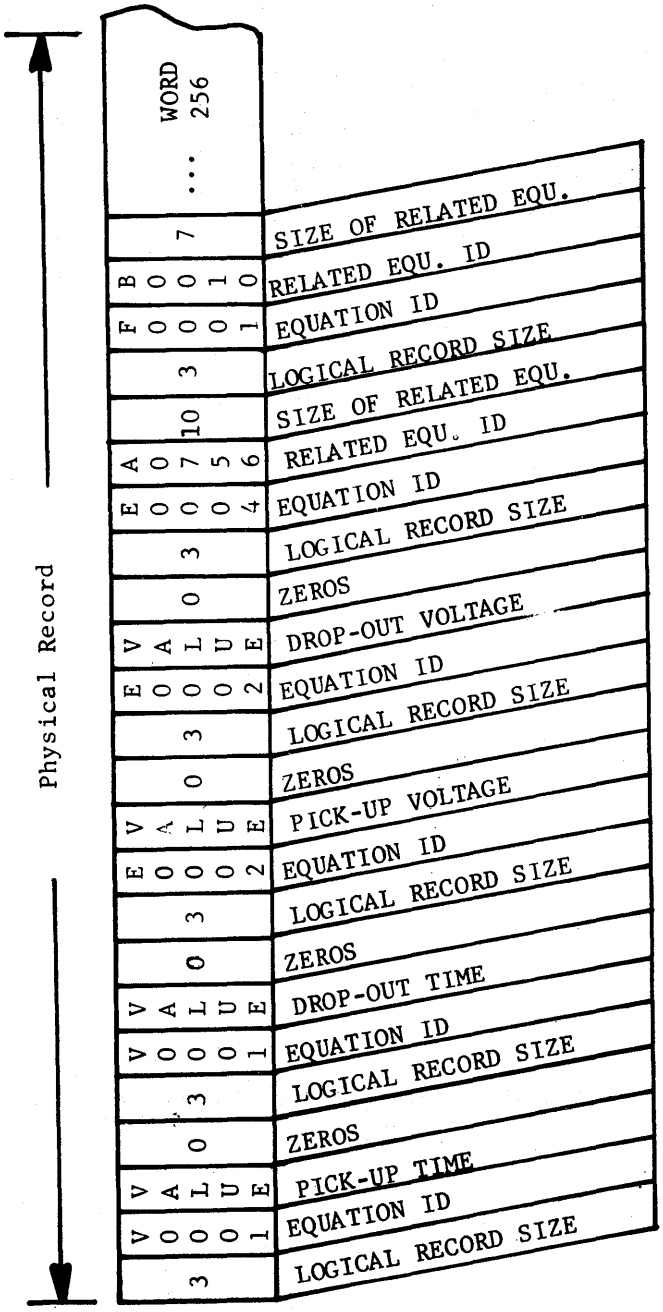


Figure 7—Cross reference file

equation cross reference information is generated and output to magnetic tape for use in later processing.

Phase 2 of the Pre-Simulation Phase completes the development of the Cross Reference File (Figure 7) and the Transfer Equation File. This is accomplished by merging the cross reference information with all related equations and generating the Cross Reference File and the Transfer Equation File of the updated Master Equation Tape (Figure 7).

All DDAS assignments are made by card input. The function of *Phase 3* is to make the DRS, Multiplexer, Frame and Channel assignments from information contained on the input cards and develop the DDAS Assignment File (Figure 8) of the Updated Master Equation Tape. This information is ordered to conform to the requirements of the Initialization Phase which develops the 4500 word commutation address table (Figure 5).

Initialization phase

The Initialization Phase performs the final processing before real-time operations. It creates the proper environment in the SDS–930 computer and on the disc storage to initiate the simulation run. The simulated switches and Discrete Outs to be input during the Real-Time Phase are also input on punched cards during the Initialization Phase. The Initialization Phase

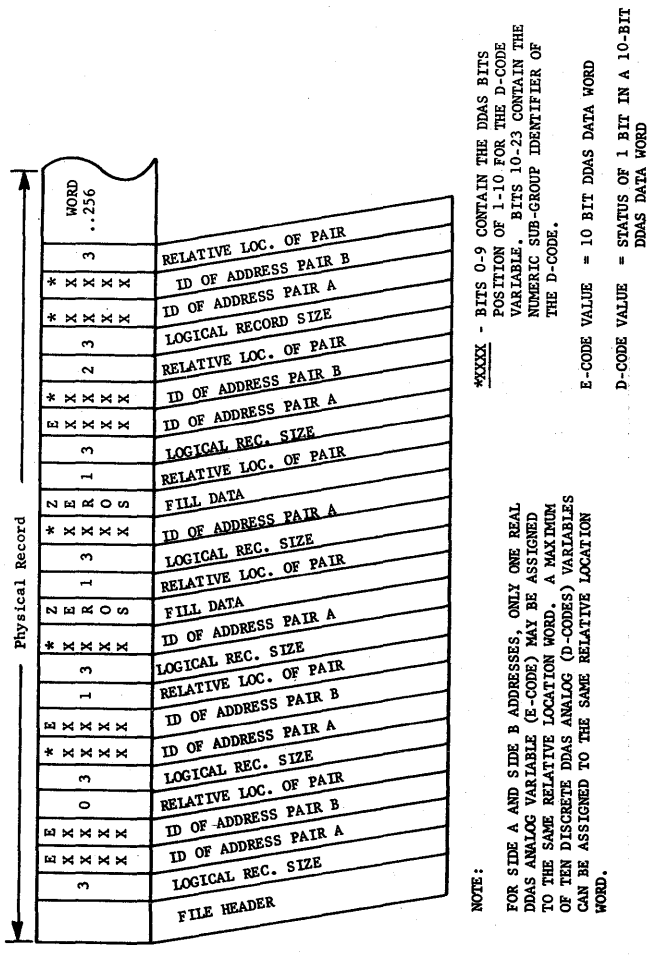
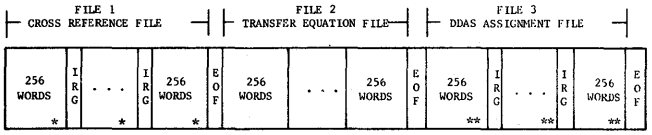


Figure 8—DDAS assignment file

utilizes these simulated variables and the Cross Reference File of the Master Equation Tape to determine which equations will be active during the simulation run. Only the active equations will be reformatted and stored on the disc (Table IV). Initialization also produces a line printer listing of the active equations and all related cross-reference information (Table V). In order to relate the simulated switches and Discrete Outs to the equations on disc during the simulation run, the Initialization Phase creates blocks of cross reference information for the switches and Discrete Outs and transfers them to the disc (Table VI). Address tables, reference tables, data tables, status tables, history buffers for recording real-time events and a 4500 word commutation table are dynamically allocated in core and initialized. The remaining available core can then be assigned as equation buffer areas. To facilitate this assignment, a list of equation sizes versus the number of equations of that size is output to the



THE THREE FILES ARE COMPOSED OF BOTH PHYSICAL AND LOGICAL RECORDS. PHYSICAL RECORDS HAVE AN ARBITRARILY SET WORD COUNT OF 256 WORDS; LOGICAL RECORDS HAVE A VARIABLE WORD COUNT CONTAINED IN THE FIRST WORD OF THE LOGICAL RECORD. EACH PHYSICAL RECORD MAY CONTAIN ONE OR MORE LOGICAL RECORDS OR ONLY A 256 WORD PORTION OF A LOGICAL RECORD (AS IN FILE 2). THUS, A LOGICAL RECORD MAY SPAN ONE OR MORE PHYSICAL RECORDS (AS IN FILE 2).

* REFERENCE Figure 7
** REFERENCE Figure 8

EOF = End of File
IRG = Inter-Record Gap

Figure 9—Master equation tape format

teletype along with the location and size of core blocks available for use as equation buffers. Utilizing the above information, the user determines the optimum number of buffers and buffer sizes and provides the information on cards for input by the Initialization Phase. All equation buffer information required by the Real-Time Phase is arranged by the Initialization Phase into the four buffer description tables (Table II).

A list of Initialization Phase capabilities is given in Table VII.

Hold phase

The Hold Phase program provides a convenient transition to real-time activities either initially or following a hold or suspension of a prior real-time simulation and gives the user the capability of controlling the environment of a specific test. He utilizes this capability by issuing commands to the Hold Phase Program via the

TABLE IV—Format of Equation on Disc

WORD	DEFINITION
0	NUMBER OF WORDS IN EQUATION RECORD (P+1)
1	DISC ADDRESS ASSIGNED THIS EQUATION RECORD
2	EQUATION MNEMONIC ID
3	REF. TO PICK-UP TIMES (REL. ADD. J FROM WORD 2)
4	REF. TO DROP-OUT TIMES. (REL. ADD K FROM WORD 2)
5	REF. TO PICK-UP VOLTAGES (REL. ADD. L FROM WORD 2)
6	REF. TO DROP-OUT VOLTAGES (REL. ADD. M FROM WORD 2)
7	REF. TO RELATED EQUA ID'S (DISC. ADDRESSES) (REL ADD.N)
8	FIRST WORD OF ACTUAL EQUATION
9	2ND WORD OF ACTUAL EQUATION
.	.
.	.
.	.
.	LAST WORD OF ACTUAL EQUATION ($ OPERATOR)
J	1ST PICK UP TIME
J+1	2ND PICK UP TIME
.	ETC.
K	1ST DROP-OUT TIME
K+1	2ND DROP-OUT TIME
.	ETC.
L	1ST PICK-UP VOLTAGE
L+1	2ND PICK-UP VOLTAGE
.	ETC.
M	1ST DROP-OUT VOLTAGE
M+1	2ND DROP-OUT VOLTAGE
.	ETC.
.	
.	
.	
N	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
N+1	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
.	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)
P	BUFFER CODE (BITS 0-4) RELATED EQ. DISC ADD. (BITS 5-23)

TABLE V—Equation Cross Reference Listing

X0000	E0001	E0002						
X0001	E0002							
X0002	E0003	F0026						
X0007	A0004							
X0008	E0004							
X0009	E0004							
X0010	F0016							
X0011	F0026							
X0012	F0026							
X0013	E0016							
X0014	F0016							
X0015	E0016							
X0016	F0017							
X0017	F0027							
X0018	E0018	E0028	F0018	F0028				
X0019	E0019							
X0020	E0019							
X0021	F0111							
X0022	F0211							
X0023	V0112	F0012						
X0024	V0112	E0029	F0012					
S0001	01110	06020	06100					
S0002	A0001							
S0009	V0011	V0012	V0112	V1000	V2222	V2500	A001	A0004
	06020	06100	D1600	D2500	D3000	D4200	D5050	E0001
	E0007	E0009	E0010	E0011	E0016	E0018	E0019	E0028
	F0026	F0027	F0028	F0111	F0211			
V0011	E0011							
V0012	A0011							
V0112	V0012							
V1000	E0006							
V2222	E0007							
V2500								
A0001	E0002							
A0004	E0005							
A0005								
A0006								
A0007								
A0011								
01110	V0012	A0005	A0007	A0011	D3000	E0007	E0011	E0018
06020	A0006	E0003	E0004					
06100	A0001	A0004	D1600	D4200	D5050	E0002	E0006	
D1600								
D2500								
D3000								
D4200								
D5050								
E0001	DV 2999.00							
	PV 4098.00							
F0111	V0011							
F0211	V0011							

teletype which initiates execution of pre-defined functions (Table VIII). He may request the current value or status of any active variable, or modify the value of status of any active variable, or performing debugging activities such as dumping any area of core or disc.

A very important option is the capability of creating a restart tape. This tape contains an image of core memory and disc memory as it was established at the end of the Initialization Phase. Thus, the restart tape provides the means by which the same or a similar real-time simulation can be executed without repeating the Initialization Phase processes. The Real-Time Simulation Phase overlays the Hold Phase in memory when the user issues the "T" directive (Table VIII).

A list of Hold Phase capabilities is given in Table IX.

TABLE VI—Switch and Cross Reference Block

WORD	DEFINITION
0	NO. OF ENTRIES (6+K+3*M)
1	BUF. CODE/DISC ADDR OF THIS BLOCK
2	NO. OF ENTRIES/REL. ADDR OF ORIG. IDS
3	NO. OF ENTRIES/REL. ADDR. OF REASSIGNED IDS
4	NO. OF ENTRIES/REL. ADDR. OF RANGES
5	/REL. ADDR. OF RELATED IDS
6	BUF. CODE/DISC ADDR OF 1ST RELATED ID
7	BUF. CODE/DISC ADDR OF 2ND RELATED ID
.	ETC.
5+K	BUF. CODE/DISC ADDR OF 1ST RELATED ID
6+K	1ST ORIG. ID
7+K	2ND ORIG. ID
.	ETC.
5+K+M	M(TH) ORIG. ID
6+K+M	START REL ADDR/END REL ADDR OF IDS RELATED TO 1ST ORIG
7+K+M	START REL ADDR/END REL ADDR OF IDS RELATED TO 2ND ORIG
.	ETC.
5+K+2M	START REL ADD/END REL ADD OF IDS RELATED TO M(TH) ORIG
6+K+2M	1ST REASSIGNED ID
7+K+2M	2ND REASSIGNED ID
.	ETC.
5+K+3M	M(TH) REASSIGNED ID

WHERE M=NO. OF ORIGINAL IDS
K=NO. OF RELATED IDS FOR ALL ORIGINAL IDS

FORMAT OF DO CROSS-REFERENCE INDEX BLOCK

WORD	DEFINITION
0	NO. OF ENTRIES (3+M+R1+R2...+RM)
1	BUFF. CODE/DISC ADDRESS OF THIS BLOCK
2	REF. TO START OF RELATED IDS (3+M)
3	NUMERIC PORTION OF 1ST DO ID/REL. ADDR. OF LAST REL. ID
4	NUMERIC PORTION OF 2ND DO ID/REL. ADDR. OF LAST REL. ID
.	ETC.
2+M	NUMERIC PORTION OF M(TH) DO/REL ADDR OF LAST REL. ID
3+M	BUF. CODE/DISC ADDR. OF 1ST DO RELATED TO DO OF WORD 3
4+M	BUF. CODE/DISC ADDR OF 2ND DO RELATED TO DO OF WORD 3
.	ETC.
2+M+R1	BUF. CODE/DISC ADDR OF R1(TH) DO RELATED TO DO OF WORD
3+M+R1	BUF. CODE/DISC ADDR OF 1ST DO RELATED TO DO OF WORD 4
4+M+R1	BUF. CODE/DISC ADDR OF 2ND DO RELATED TO DO OF WORD 4
	ETC.

Real-time simulation phase

The Real-Time Simulation Phase drives the SDS-930 computer and the SDS Interface (Figure 2) for the actual simulation of the launch vehicle complex. Operation during the Real-Time Phase also requires the execution of the launch computer programs, operating

TABLE VII—Initialization Phase Capabilities

- Extracts the specific equations to be active from the cross-reference file of the Master Equation Tape utilizing information input from cards.
- Prints a cross-reference listing of all variables to be active.
- Outputs a summary of number of equations versus equation length.
- Outputs the location and size of core blocks available for use as equation buffers.
- Assigns a disc address to each active equation.
- Reformats each active equation and its cross-reference data and stores the combined results on disc.
- Prints a listing of all equations stored on the disc.
- Creates cross-reference data for all active DO's and switches in a format usable in real-time. This is also stored on the disc.
- Assigns core locations for all necessary tables.
- Assigns core locations to be used as equation input buffers.
- Initializes all tables and buffers.

TABLE VIII—Hold Phase Commands

Hold Phase directives (commands) and options requested are as follows:

DIRECTIVE CHARACTER	OPTION REQUESTED
I	Print Hold Phase Instructions
N	Write Memory on New Restart Tape
R	Restore Memory from Previous Tape
P	Proceed to Post-Simulation Phase
T	Proceed to Real-Time Simulation Phase
F	Print F-Type Data Table
E	Print E-Type Data Table
U	Update Data Value or Update Status
Q	Dump Disc on Line Printer
V	Write Memory and Disc on Restart Tape
D	Restore Memory and Disc from Restart Tape
S	Print Summary of True Status
B	Branch to 32K Debug
C	Exit Debug

asynchronously, in the RCA–110A computers. Collectively, the three computers, their operating software, the interface hardware, and the launch complex/vehicle mathematical model provide the principal ingredients for the simulation of a Saturn V launch vehicle complex (Figure 2).

The simulation process consists primarily of: (1) equation evaluation triggered by inputs from the launch computers by internally generated values and from the card reader by manual input, and (2) outputs developed as a result of equation evaluations which are made available to the SDS interface for use by the launch computers.

The communication between the launch computers (RCA–110A) and the Simulator (SDS–930) is through the SDS Interface. This communication is in the form of discrete signals sent from either RCA–110A to the SDS–930 (Discrete Out or DO), discrete signals requested by either RCA–110A from the SDS–930 memory (Discrete In or DI) and DDAS analog and discrete signals available at the SDS Interface for the RCA–110A computers to access when desired. Simulated DO's and switch settings can be input to the Real-Time Simulation Phase from the card reader by operator action.

Upon receipt of a Discrete Output from the SDS Interface, the Real-Time Simulation Phase will schedule a chain of Boolean equations for evaluation. Some equations must be evaluated when there is a change in status of a dependent variable, some other equations must be evaluated at regular time intervals or at the end of a set time period, and still others must be evaluated when a specific analog variable reaches a particular value.

Each equation that is to be evaluated must be retrieved from the disc where it was stored during the Initialization Phase. Each equation record contains information required for evaluation such as pick-up and drop-out times, pick-up and drop-out values, and equation identifiers of related equations. The identifier or ID of an equation is the name used to refer to the dependent variable on the left side of an equation.

In general, each DO received from the SDS Interface will require an equation evaluation which in turn generates a response (DI) to be supplied to the SDS Interface which makes the response available to the RCA–110A computers. During equation evaluation DDAS responses are also developed and provided to the SDS Interface for DDAS commutation to the RCA–110A computers. The generation of a response may require the evaluation of several dependent equations. An example should explain the evaluation process.[12] We assume the following three logical equations are given:

$$Y_1 = Y_3 + \mathrm{DO} * Y_1 + \mathrm{DO} * Y_2$$

$$Y_2 = \mathrm{DO} * Y_1 * Y_3 + Y_1 * \overline{\mathrm{DO}} + Y_2 * Y_1$$

$$Y_3 = \bar{Y}_1 * \overline{\mathrm{DO}} * Y_2 + \bar{Y}_1 * \mathrm{DO} * \bar{Y}_2 + \bar{Y}_1 * Y_3 + Y_3 * \mathrm{DO}$$

The three variables Y_1, Y_2, and Y_3 appear on the left side of the equations and also on the right side of the equations. If the external variable DO changes state from "1" to "0," the first evaluation of the three equations yields the intermediate result $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 1$. The next evaluation of the same equations using these intermediate results yields the new values $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 0$. This process of successive evaluation is continued until a stable state, i.e., no change of state of any variable, is reached. In this example, four evaluations have to be performed. If no stable state can be reached, the circuit oscillates. This case occurs in the example if another initial condition is used where the third evaluation yields the same initial state. Figure 6a illustrates the evaluation process and Figure 6b depicts all possible states of the example.

As each equation evaluation is completed, the necessary information with which to schedule the evaluation of all affected or related equations must be immediately available. This is done by having all related equation ID's available as each equation is brought in from disc and by logging all related equation ID's into the appropriate queue.

It is obvious from this example that the processing of many equations within the Real-Time Phase can become time-critical.

Throughout the Real-Time Phase execution, a complete record of events is recorded on the History Tape. This magnetic tape contains information such as equation ID's, times at which a state changed; and in the

TABLE IX—Hold Phase Capabilities

- Record memory in an initialized state on a magnetic tape.
- Restore memory to an initialized state from a previously created tape.
- Record memory in an initialized state and the disc contents on a magnetic tape.
- Restore memory and the disc contents to an initialized state from a previously created tape.
- Print the status of any variable within the system.
- Change the status of any variable within the system.
- Advance the simulator to the real-time phase.
- Advance the simulator to the post processing phase.
- Utilize the capabilities of the SDS Program, 32K DEBUG.

TABLE X—Real-Time Phase Capabilities

- Responds to DO inputs from the RCA-110A's.
- Responds to SWITCH inputs from the card reader.
- Responds to simulated DO inputs from the card reader.
- Responds to system interrupts--millisecond clock, elapsed timer and system error interrupts.
- Responds to disc interrupts.
- Evaluates appropriate boolean equations.
- Maintains tables of all conditions of the system.
- Maintains clocks for various timing processes.
- Records all variable changes and evaluation results of the system on magnetic tape.
- Terminates real-time functions upon command when the system becomes stabilized.

case of analogs, the present and previous value of the analog variable at a given time. This tape is used as input to the Post-Simulation Phase to record on the printer the complete history of the particular simulation run.

A list of capabilities of the Real-Time Phase is shown in Table X and a simplified diagram of the overall processing is shown in Figure 10.

Post-simulation phase

The Post-Simulation Phase processes the Real-Time Phase history tape according to options selected from the SDS–930 console. The test conductor is provided the following options he can choose via the switches.

1. A listing of all events on the history tape.
2. A listing of only events where a change in status or value occurred.
3. A line printer plot of up to eight analog variables (analog value versus time).

The user also specifies (via card input) which system variables (1 to 10) he desires to be printed and the relative time of the first and last event to be printed (nominally 0 to 24 hours). This card input is then used in conjunction with options (1) or (2) above to provide the particular listing desired.

A complete list of Post-Simulation Capabilities is shown in Table XI.

SIMULATOR DIAGNOSTICS

The functional design of the acceptance and diagnostic program not only provides a method for thorough checkout of the SDS-designed interface equipment, but also results in an effective and permanent system of diagnostic procedures. Modularity, simplicity, and thoroughness comprise the basic philosophy used in the design of the acceptance test procedures. To simplify system programming problems, the acceptance test and diagnostic programs function within the framework of the standard capabilities provided by the RCA–110A TAME System (Assembler, Library Routines, Utility Routines, and Loader).

The order in which the diagnostic tests are performed follow the order as outlined in the design specification.[8] This is to insure that certain interface equipment components are working correctly before that component is used in another test. The functional design of the system, however, provides complete test independence. Therefore, the order of tests may be selected at random to facilitate rapid location of a suspected malfunction.

Since the diagnostic programs operate within the RCA–110A, the standard RCA–110A mainframe diagnostics are used to insure that the RCA–110A mainframe, I/O data trunks, and all peripheral equipment (excluding the SDS Interface) are operating properly.

SDS-930—RCA-110A communications

To effectively control and coordinate the tests and diagnostic programs, the RCA–110A issues command words (Discrete Outputs) which consist of a predefined octal bit configuration to the SDS–930 via the Discrete Output Channel. Once the command word is received by the SDS–930, it is interrogated to determine which test should be initialized. When the interrogation procedure is satisfied, a response word is sent back to the RCA–110A via the Discrete Input Channel.

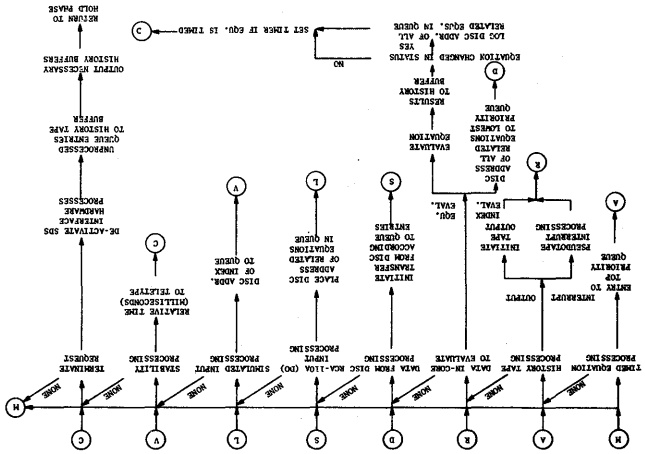


Figure 10—Simulator flow diagram

Test control loop

The Test Control Loop is the system monitor which controls the scheduling of tests. It monitors the various sense switches and transfers control to the requested diagnostic subprogram when the appropriate switch is toggled. The title of the diagnostic is printed at the beginning of each test.

Each diagnostic subprogram has the ability of printing a list of Predicted versus Received data (PRED/RVCD) for each error. This option is selected during monitor control by toggling a sense switch on the RCA–110A control panel. It causes the diagnostic subprogram to list all interface data errors (predicted value and actual erroneous values received) encountered by the data validity checks performed within each diagnostic subprogram. Control may be transferred to the monitor at any time by setting a priority request switch.

An option for dumping raw data exactly as it was received from the SDS Interface is available to the user by setting a sense switch before each diagnostic test is executed. Another sense switch assignment permits automatic looping of a test. This option becomes particularly useful when the engineer is isolating a suspected malfunction and he wishes to continuously maintain a programmed loop to observe a data input or output signal to or from the SDS Interface.

The *Discrete Output Diagnostic* is designed to insure the communication link between the RCA–110A and SDS–930 computers via the Discrete Out (DO) data channel is operating properly. Since the DO data channel is used for systems test control, the validity of this link must be tested before any other tests are conducted. Five unique bit configurations are transmitted via the DO data channel to the SDS–930 and are immediately strobed back into the RCA–110A. If each bit configuration returns exactly as transmitted, the DO data link is functioning properly and the communication link is established.

The *Discrete Input Diagnostic* checks the two Discrete In (DI) data channels between RCA–110A and SDS–930 computers. All modes of DI scan are executed by transmitting a unique bit configuration to the SDS–930 via the DO data channel followed by the DI scan which gathers the same data back into RCA–110A core memory (Status and Log Tables). Each bit configuration transmitted is compared to each bit configuration received. Data channels are also checked for parity error, inoperative, etc.

The *Multiple Operand Address Diagnostic* (*MTOAD*) assures that the addressing of more than one group of DO lines will cause an error condition, and automatically inhibit further DO transmission. The test consists of generating successive pairs of illegal addresses, e.g., (1–2, 2–3, etc.) until all addresses have been made. Between each MTOAD condition, an attempt to transmit a legal address is made as a check on the automatic inhibit caused by the MTOAD. When the legal address fails transmission, the interface is reset and the next successive MTOAD is generated.

The time signals used with the discrete signals include those of the Eastern Standard Time and the Relative Time Counter. The *Relative Timer Diagnostic* checks the bit incrementation of the Relative Time Counter (27 bit, 1 ms). This counter supplies the means for having a zero time reference in the computer which can be reset under program control.

The EST Timer Diagnostic checks for proper setting and incrementation of the Gray Code counter. Preset Gray Code values are sent to the SDS Interface via the DO data channel. If the preset value is not returned, or is returned in error, two more attempts are made. After the third unsuccessful attempt, the preset value and the returned value (if any) are printed as an error message. Upon detection of the proper preset value, the RCA–110A checks for proper incrementation. Any errors detected are logged with the expected time and the received time.

The *Elapsed Timers Diagnostic* is initiated by the RCA–110A but the main functions of the diagnostic checks are performed within the SDS–930. The elapsed timers consist of 190 *fixed* consecutive memory cells. After the first timer has counted to zero, an interrupt occurs, whereupon the elapsed timers are interrogated and compared to timing criteria within the discrete and DDAS activities.

The first part of the *Digital Data Acquisition System DDAS Diagnostic* assures that commutation occurs at a 3.6 KC rate. This test is initiated by the RCA–110A but is performed by the SDS–930 program which cycles through and updates each DDAS data word once every

TABLE XI—Post-Simulation Phase Capabilities

- Time span - time to begin and end printing
- Variables to be listed

Only the changes in value or status of variables on the history tape will be printed unless the test conductor specifies that all information be printed by setting a "BREAKPOINT" on the SDS-930 console.

A secondary function of the post processor is to provide a line printer plot of any analog variable within the system. The value of the analog variable is plotted against time. Again, the test conductor is allowed to specify the time span and the analog variable(s) to be plotted.

250 ms. After the commutation test is completed, the RCA–110A starts the *DDAS Interface Test,* which consists of exercising the DDAS Interface to insure proper commutation on all Digital Receiving Stations (DRS), verifying valid data and time responses for all modes and submodes of DDAS scans.

The *ACE Interface Diagnostic* checks for proper transfer of data words between the RCA–110A and the SDS–930 computers via the ACE equipment by comparing four data words with different bit patterns.

Total dynamic system test

This test is designed to exercise all interface equipment between both RCA–110A computers and the SDS–930 computer. The programs are executed such that peak data transmission operations occur between the three computers.

Either of the two RCA–110A computers may initiate the test by transmitting the appropriate command word. The SDS–930 computer initializes its discrete system, commutation is started on all five Digital Receiving Stations, and the Eastern Standard Clock is preset to zero. All three computers (both RCA–110A's and SDS–930) record the type and number of system errors detected, total number of discrete transmissions, and the time duration of the test.

CONCLUSIONS

The simulator was successfully demonstrated in January 1969 with several RCA–110A test programs for the Instrumentation Unit (IU). The IU was used because it is the most complex system of the Saturn V, and it was determined that, if it could be simulated, then the other stages could be also.

The checkout of the software for all timing and logic combinations was extremely difficult within a real-time computer network and the special hardware interface. Often hardware and software errors occurred simultaneously obscuring the source of the error.

The successful simulation runs of the IU test programs have proven that the concept of the real-time digital simulation for test program evaluation is feasible. Though the determination of all timing limitations with respect to size of data base and minimum component time-constants is still subject to further study, it is established that the simulator can be used in many applications without changes in the software. Systems which predominantly contain devices with relatively large time constants such as electro-mechanical, mechanical, and pneumatic devices are particularly suited for this simulator. Fast electronic logic circuits can also be simulated if they control other slower devices so that the minimum time between stimuli output and response input is in the range of milliseconds to second.

The software allows easy change of parameters, of tolerances, and of the configuration of the hardware under test for studying its effects on the overall systems performance, for evaluating the completeness of test programs, and for locating malfunctions. Hence, the real-time software simulator can be used for applications such as test-program design and evaluation, malfunction analysis, hardware design analysis and training of checkout and launch personnel.

The data base can be set up easily since translation of the schematics into the logical/analog equations is not difficult. However, the data base should be established while the hardware systems are being designed so that the design engineers are modeling their own design and thus assure the establishment of a data base which accurately reflects the hardware design. Also, the simulator can then be utilized as an active tool during the design phase and can be used for early test program design and evaluation before hardware delivery.

It is also conceivable that the logical/analog equations may describe the functions of subsystems on a higher level than the piece-part level and thus very large systems could be simulated to less depth.

Space missions of great complexity and of longer duration, and a greater frequency of launchings will require speeding up of checkout operations and a constant thorough knowledge of the status of all systems within the space vehicle. The vehicles themselves will be more complex and their configuration of greater variety which will result in an increase of design of new and more sophisticated test procedures. Narrow launch windows and more frequent launchings will result in the need for short turnaround time at the launch pad. In order to meet this challenge in new space programs such as the Space Shuttle and Space Station, plans are being made as to how this powerful simulation system can be optimally used for these new programs. Primarily, hardware changes for the interfaces have to be identified in order to make them more general. The software changes of the actual simulation program are expected to be minor; however, it might be necessary to write some data formatting routines for the interfaces.

ACKNOWLEDGMENTS

The authors are indebted to their colleagues at Marshall Space Flight Center who were part of the project team.

In particular, R. G. Abe, presently at Computer Sciences Corporation's Western Region, who detailed, designed and implemented the major portion of the simulation software, is fully acknowledged for his outstanding efforts. Credit is gratefully given to T. L. Balentine for his development of the diagnostic software, and to E. E. Branstetter and R. Hull for their programming contributions. The authors wish to specifically thank their colleagues at Astrionics Laboratory who provided the hardware systems operations and the data base.

REFERENCES

1 *Saturn V system development breadboard facility data plan*
Document Number D5-15207 NASA Contract NAS8-5608
The Boeing Company
January 1965

2 *Saturn V system development breadboard facility operational plan*
Document Number D5-15201 NASA Contract NAS8-5608
The Boeing Company
November 1964

3 *ESE simulation*
NASA Contract NASw-410 May 1965
General Electric Company

4 R L JAEGLY
Test procedure validation by computer simulation
Flight test Simulation and Support Conference AIAA Paper Number 69-280 March 1968

5 L W BROOKS J A STAHLEY
Real-time digital simulation of the Saturn V system
Sperry Rand Engineering Review Systems-Computer Applications 1969

6 *VLF-39-1 digital data acquisition system (DDAS) for Saturn V*
Interim Technical Report
Marshall Space Flight Center Astrionics Laboratory

7 L W BROOKS L J GELLMAN J A STAHLEY
Transfer equation preparation manual for launch vehicle and ground support equipment system simulator
Prepared for NASA Marshall Space Flight Center (R-ASTR-ESA)
Sperry Rand Corporation Huntsville Alabama 1967

8 T L BALENTINE C O RIGBY R G ABE
Design documentation for RCA-110A computer programs for DEE-6D modification acceptance tests
Computer Sciences Corporation Huntsville Alabama December 19 1967

9 T L BALENTINE
Operating procedures for RCA-110A computer programs for DEE-6D modification acceptance tests
Computer Sciences Corportion Huntsville Alabama June 20 1968

10 *Acceptance test procedure—DEE-6 simulator system*
Scientific Data Systems Document Number SDS-144934
NASA Contract NAS-11809 May 1967

11 L W BROOKS C L McCUBBINS
Integrated test plan for acceptance of DEE-6D modification
Sperry Rand Corporation Huntsville Alabama June 1967

12 S H CALDWELL
Switching circuits and logical design
p 468 John Wiley & Sons New York

13 R G ABE K SCARBOROUGH
General design specification for launch vehicle and ground support equipment simulator system
Computer Sciences Corporation Huntsville Alabama March 1967

14 *Program specifications for launch vehicle and ground support equipment simulator system*
Computer Sciences Corporation Huntsville Alabama To be published in spring of 1970

Picturelab—An interactive facility for experimentation in picture processing

by E. ARTHURS, W. S. BARTLETT, D. J. LADD, R. L. SALMON and J. H. WHIPPLE

Bell Telephone Laboratories, Incorporated
Whippany, New Jersey

INTRODUCTION

Picturelab is an interactive program system for experimentation in picture processing built using the SIS facility[1] on a GE 635. Picturelab has been prepared as a replacement for a previous batch facility. The batch facility had the major drawback that most studies had to be run overnight. When graphic results were required microfilm processing time was added to the already long turn-around time. Since most present research in image processing continues to be done on a trial-and-error basis, the long turn-around times associated with the batch runs resulted in a combination of painfully slow progress and long running times.

In place of this, Picturelab permits interactive processing and immediate viewing of the processing results. This approach permits faster progress because of continuity of attention, quick elimination of fruitless paths, and capacity to test a much greater variety of algorithms in a given period of time. The material following is organized into a section introducing digital representation of pictures and sections discussing design considerations and outlining the basic portions of the Picturelab system. The appendix contains an example of an interactive session.

BRIEF INTRODUCTION TO DIGITAL REPRESENTATION OF PICTURE INFORMATION

Picture information is represented digitally by subdividing a picture into a rectangular array of points, and storing the level of grayness of each point as a binary integer. The gray level value is generally obtained in one of two ways, depending upon the storage medium. In the case of 35mm transparencies, the gray level is obtained by shining a spot of light through the transparency and measuring the amount of light transmitted. Since the spot of light must be moved about the picture, a conventional technique is to locate the picture in front of a Cathode Ray Tube and use beam positioning to control spot placement. The spot is made to move regularly along the rows and the amount of light that is transmitted is converted to a digital value and stored on magnetic tape. This type of scanner is referred to as a "flying spot scanner".

Picture information stored on opaque paper may be scanned by a facsimile device. In a facsimile scanner the material is wrapped around a drum and the drum is rotated under a spot of light. The reflectance is measured and converted into digital form and stored. During rotation the spot is made to move down along the length of paper.

To give an example of the volume of data produced, a 35mm slide is subdivided into approximately 1000 rows and 1000 columns. A five-bit gray-level value is stored for each of the 1 million row-column intersection points, thus producing 5 million bits of information. The regular two-dimensional array of intensity information is conventionally referred to as a *raster* or *gray-scale* or *half-tone* representation. Such a representation is utilized for present day television pictures.

DESIGN CONSIDERATIONS

The design criteria set forth for the Picturelab system were that the system should be (1) interactive, (2) command driven, (3) permit symbolic variable referencing, (4) permit the creation and execution of symbolic command files, (5) permit storing of partially processed pictures, and (6) be usable not only interactively but also as a batch program. In addition, the commands used for batch processing should be the same as the commands used for interaction whenever possible. Finally, of course, the arrays of picture information

that could be processed were to be as large as possible. All of these criteria were actually achieved including the capacity to handle picture fragments up to 128 x 128 points.

The reason for making the system interactive was that, as previously mentioned, most picture processing research proceeds in a trial-and-error manner. A suitably implemented interactive system permits continuity of attention, quick elimination of fruitless paths, and faster progress down fruitful paths.

The decision to make the system command driven was based upon the desire to permit interaction to proceed quickly, to permit symbolic files of commands, and to permit commands punched on cards for batch input to be of exactly the same form as the commands used at the console.

The provision for symbolic variable referencing was founded on the desire to make the use of the commands as much as possible like programming in conventional high level languages (e.g., FORTRAN). One of the basic features of all programming languages is the use of symbolic names for variable information. This is accomplished in Picturelab by providing the user with a symbol table in which he may store variable names and associated values. He may then access the values for use in commands simply by supplying the variable name.

The provision for creating files of commands permits accumulating together, under a single name, a number of commands required to control a single activity. The single name may be used in place of the sequence of commands in other contexts. This is the familiar subroutine concept of higher level languages.

A major demand to be made upon the system was that of access to picture information. In order to permit efficient execution is was decided that information being processed should be held in core storage. It was not possible to permit holding in core an entire picture. Rather, portions of pictures called *fragments* are selected from within a picture for processing and placed within a system *Data Area*. The information that is accessed by processing programs is whatever picture fragment is stored in the Data Area at the time of access.

Another decision made about the storage of picture information was that, for a given picture fragment in the Data Area, the data should be stored with one point per word. Information from different pictures (raw and processed data, for example) is stored in different sequences of bits within the words. The two-dimensional array of bits obtained by taking the same sequence of bits within all the words used for a picture fragment is called a picture *Plane*. Using Planes permits optimizing the program that does the packing and unpacking of the data from a specific set of bits. This program can be made to execute faster than one unpacking data from varying sets of bits as would be required if many points from a single picture were stored within one word.

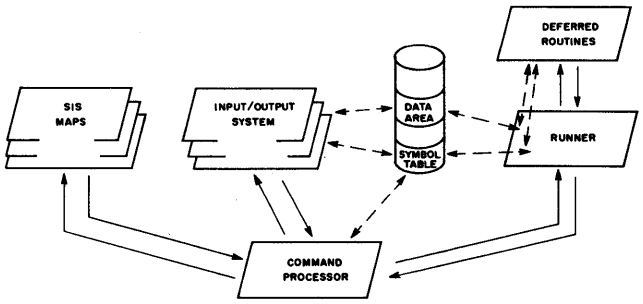


Figure 1

The size picture fragment that can be accommodated is directly related to the amount of storage available for the Data Area. It was decided that 18,000 words (enough to hold a 128 x 128 fragment) would be an acceptable compromise between the demand for larger fragments and the desire to conserve core storage.

The programs (called *Routines*) which implement the picture processing algorithms operate on whatever data happens to be in the Data Area at the time of execution. In general, the arguments to a Routine will indicate one Plane from which to obtain information and another Plane in which to store the results of the processing.

THE PICTURELAB SYSTEM

The fundamental structure of the system consists of two basic data tables and three basic program systems. See Figure 1.

The data tables are the system Symbol Table for holding variable names and their associated values and the system Data Area which holds the current picture fragment. When not in core these tables are maintained in a file system in addition to all of the processing Routines, and saved sequences of commands (called *Processes*). The three program subsystems and their responsibilities are listed below.

Subsystem	*Purpose*
Command Processor	Create and maintain the Symbol Table, Initiate execution of Routines, Create and maintain Processes.
I/O System	Read picture information into Data Area from Tape or Disc,

	Store picture information from Data Area on Tape or Disc.
Library of Routines	Perform picture processing algorithms.

The Routines that actually perform the picture processing are written in FORTRAN and are compiled exterior to the Picturelab system. These Routines are then converted into executable form by the GE system loader and stored within the file system as executable core images. Commanding the execution of a Routine results in its being loaded along with the Data Area and control being passed to it. At the completion of execution, control is returned to the Command Processor for communication with the user to obtain his next command.

GRAPHIC CAPABILITIES

Since human comprehension of pictures depends on pictorial information rather than lists of data, it was felt mandatory that the user should be able, during a console session, to view the picture information stored within the Data Area. This is accomplished through use of the SIGHT[2] console. The console consists of a Digital Equipment Corporation PDP–7 computer and a Cathode Ray Tube display. The display has 1024 x 1024 addressable points and a spot may be illuminated at one of eight levels of intensity. The intensity modulation permits direct viewing of half-tone (gray-scale) pictures on the scope face. The PDP–7 contains 8192 words of core storage. Picture information for display is held in core storage in the form of scope vector commands. The commands draw horizontal vectors of constant intensity. Changes in intensity are accomplished by starting a new vector with a new intensity.

The size picture that may be stored depends upon the frequency of changes of intensity. More than one picture may be stored, if space permits, and selection of the picture to be displayed is accomplished by means of function buttons. Sample capabilities are one half-tone picture or three or four binary (two level: black and white) pictures. Figure 2 is a photograph of a half-tone display.

In addition to the interactive graphic capability provision is included for the generation of microfilm from binary pictures using a Stromberg Datagraphix 4060 Microfilm Recorder. The information plotted on one frame may include data from a number of picture fragments, thus permitting an entire one million point binary picture to be plotted on a single frame. In effect this creates a 35mm transparency of the same size as the original.

INTERESTING IMPLEMENTATION APPROACHES

The combined constraints of needing to work within a relatively small amount of core storage, desiring to have execution proceed as quickly as possible and desiring to do most of the coding in Fortran led us to utilizing a number of sophisticated approaches which may be of interest to others attempting similar systems.

The approaches to be discussed here include (1) swapping of the SIS executive and a special command structure to improve the speed when this is done, (2) dynamic modification of subroutine calling sequences to improve execution time, (3) recursive overlaying of the entire Command Processor in order to implement a block structure capability for the Processes, and (4) inclusion of an arithmetic statement compiler and executor.

SIS,[1] the system upon which Picturelab is built, normally executes in 32K words of core storage. This area is subdivided into about 20K for the SIS executive itself, and 12K for user programs, called Maps, that execute under control of the executive. In order to accommodate up to 18,000 words of picture data it was decided to swap the SIS executive, which provides a file system and a device-independent I/O system, out of core and utilize the freed space for the picture information.

This necessitated the writing of a minimal (3K) executive to control algorithm execution. The executive provides restricted I/O capability and can access only files for which it has absolute disk addresses. This executive (called the *Runner*) is called by the Command

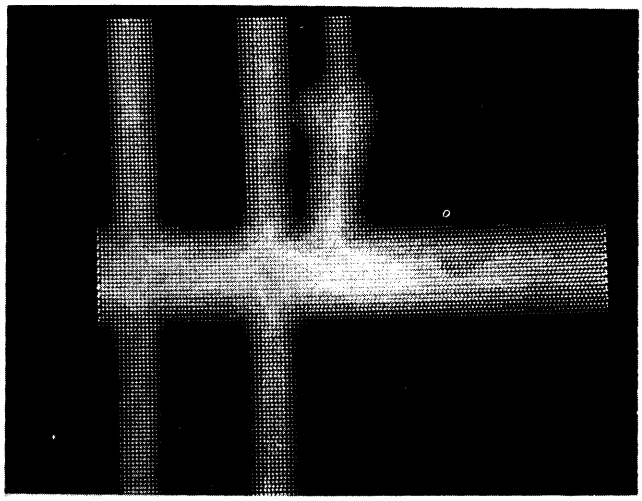


Figure 2—Enlargement of a portion of a raster picture

Processor when the execution of a Routine is requested. It swaps the SIS executive, loads the Routine to do the actual processing, and after its completion restores the SIS executive and returns control to the Command Processor.

In order to increase the speed at which the execution of routines could proceed a special mode (called *Deferred Execution*) was provided which permits accumulation of a series of Routine references and, on command, executes the entire list, swapping the SIS executive out at the beginning and restoring it only after completion of execution of the entire list of Routines.

The second approach, dynamic modification of subroutine calling sequences, speeds up the unpacking of data. This arises during the execution of algorithms on picture fragments; the desired picture information must be unpacked from a picture Plane within the word operated upon, and then restored to another Plane. It was desired to make this unpacking as fast as possible, but also it was desired to permit the Routines to be written in Fortran. In Fortran most bitpicking operations, such as unpacking data from within a word, are slow. One alternative frequently taken is to use subroutines to perform these functions; however, frequently the subroutine linkage can take more time for such routines than the actual processing instructions. The approach taken in Picturelab is to provide bit manipulating subroutines, which, on first execution, overwrite the calling sequence with the instructions to be executed and then execute them. Subsequent passages through the subroutine reference result in immediate in-line execution rather than a subroutine call. Naturally, this works only when the variability in subroutine argument values are suitably restricted.

Another approach of potential interest is that of the method used to permit local symbols within Processes. In order to avoid symbol clashes between symbols used in different Processes (recall that these are files of sequences of commands to the Command Processor), it was desirable to permit symbols local to a Process and the capability to hand down symbols not redefined within a Process.

There is an inherently recursive structure to the procedure of unwinding these Processes, however, Fortran is not oriented toward recursive processing. The solution was to take advantage of the stacking capability of SIS. A Map, executing under control of SIS may ask for another Map to be executed, during which time the entire core image of calling Map is placed on top of a Map stack. Upon completion of execution of the called Map its core image is destroyed and the calling Map restored. This restoration process is called *peeling back*. A Map may call itself, resulting in the placement of the existing core image on the Map stack and the loading of a fresh core image. This may be repeated to any (reasonable) depth.

During the execution of a Process, each time the Command Processor encounters a reference to another Process, it simply writes out the Symbol Table and a small amount of other bookkeeping data, stacks itself and calls in a new image which loads the Symbol Table and data; adds new local symbols to the end of the Symbol Table; and executes the called process. Upon completion of the execution of a Process, the Command Processor removes the local symbols, saves the Symbol Table and bookkeeping data and peels back to the previous version of itself. This approach permits an inherently recursive structure to be handled by a relatively simple Fortran Program.

A final implementation item of potential interest is an immediate execution arithmetic statement processing Routine. All Routines, the programs which operate on picture information in the Data Area, must be compiled outside the SIS system and loaded into the system as absolute core images by means of an off-line run. To permit flexibility during a console session in spite of this requirement, a routine was written which will dynamically compile single assignment statements. These statements permit both logical and arithmetic operators. For example $A \leftarrow (B > C). D + (B < = C). (-D)$ places the contents of D into A if B is greater than C otherwise $-D$ is placed in A.

Of particular note is the fact that if the variable on the left of the assignment arrow refers to a picture Plane, the result is stored in all the points within the picture; if a Plane is mentioned to the right of the arrow the execution utilizes the corresponding picture data for all the points in the picture in the execution of the statement. This capability coupled with the capacity to loop within a Process provides the system user with a very powerful on-line programming capability for processing picture data. This routine provides a happy compromise between the flexibility of an interpreter and the speed of executable code.

SUMMARY

To summarize, Picturelab is an interactive system intended for use for research in processing digitized pictures. The system contains a Symbol Table for symbolically handling variable information, a Data Area for in-core storage of pictures to be processed, and provision for files containing lists of symbolic commands.

System action is directed by commands entered into a Command Processor. These commands manipulate the Symbol Table, call into execution picture processing Routines, or manipulate files of commands. In addition,

the user may gain access to data on secondary storage through an I/O system which permits reading picture information into and out of the Data Area.

Visual interaction with the picture information in the Data Area is accomplished by means of the SIGHT console and Stromberg Datagraphix Microfilm output.

ACKNOWLEDGMENTS

The Picturelab system is a result of the effort of a number of people. Not included in the list of authors are Mr. K. J. Busch who contributed substantially to the design and Miss K. M. Keller who implemented most of the Command Processor.

REFERENCES

1 K J BUSCH G W R LUDERER
The slave interactive system: A one-user interactive executive grafted on a remote-batch computing system
Interactive Systems for Experimental Applied Mathematics pp 225–240 Academic Press Inc 1968

2 W S BARTLETT K J BUSCH M L FLYNN R L SALMON
SIGHT—A Sattelite Interactive Graphic Terminal
Proceedings of 23rd National ACM Conference Brandon Systems Press Inc

APPENDIX

Sample picturelab session

This Appendix presents a sample of the use of Picturelab commands. The operations described below are shown in Table 1 at the end of the Appendix. Lines preceded by a star were printed by the system.

A. Three planes are defined to be global variables. The numbers give the starting bit and number of bits within a 36-bit word. The masks defined are then listed.
B. Control is passed to the I/O system, the Data Area cleared, an input file defined, and the label on the file is printed. A picture fragment of size 128 x 128 starting at row 641, column 636 is requested. The picture is read and control returned to the Command processor.
C. A routine applying a constant threshold to the picture in plane INPLAN is executed with the results going into plane OUTPLA. Then the original picture is displayed followed by the threshold results.
D. A Routine applying an adaptive threshold is executed and the results displayed.
E. Control is passed to the I/O system in order to save the picture on a file, SAVE THRESH, labeled "Results of Thresholding". The values saved are from the plane OUTPLA created by the adaptive thresholding routine.
F. Then a Process is built containing commands to perform an adaptive threshold and then display the results. This is filed under the name ADTHRE.
G. This Process is then executed with a tracing capability enabled to permit monitoring the flow of control.

TABLE I—Sample Cønsøle Sessiøn

```
* PICTURELAB. CØMMAND PRØCESSØR. TYPE CØMMAND, MAP, RØUTINE ØR PRØCESS NAME
      GLØBAL INPLAN PLANE  31 6
      GLØBAL ØUTPLA PLANE  15 1
      GLØBAL PLDISP PLANE  32 3                    A. DEFINE PLANES
      PRVAL INPLAN ØUTPLA PLDISP
* INPLAN PLANE    31  6  000000000077
* ØUTPLA PLANE    15  1  000010000000
* PLDISP PLANE    32  3  000000000034
      IØSYS
* I/Ø SYSTEM
      CLEARD
      INPUT 21  1                                  B. ENTER I/Ø SYSTEM,
* ENGINEERING DRAWING #1 FRØM MH8361                  READ PICTURE FRAGMENT
      WINDØW  128  128
      SUBPIC   641  636
      RDPCT INPLAN
      PEEL
* CØMMAND PRØCESSØR.
      THRESH INPLAN ØUTPLA  15                     C. APPLY CØNSTANT THRESHØLD,
      DISPLA PLDISP                                   DISPLAY
      DISPLA ØUTPLA
      ADAPT INPLAN ØUTPLA                          D. ADAPTIVE THRESHØLD
      DISPLA ØUTPLA
      IØSYS
* I/Ø SYSTEM                                        E. SAVE ØUTPUT ØF ADAPTIVE THRESHØLD
      ØLABEL "RESULTS ØF THRESHOLDING"
      ØUTPUT SAVE THRESH
      WRPCT ØUTPLA
      PEEL
* CØMMAND PRØCESSØR.
      BUILD                                        F. BUILD A PRØCESS TØ DØ ADAPTIVE THRESHØLD
      PARS PLIN PLØUT
   10 ADAPT PLIN PLØUT
   20 DISPLA PLØUT
      FILE ADTHRE
      TRACE ØN
      ADTHRE INPLAN ØUTPLA                         G. EXECUTE PRØCESS
* ENTERING ADAPT
* ENTERING DISPLA
* CØMMAND PRØCESSØR
      PEEL
      BYE                                          H. TERMINATE
```

BELL TELEPHONE LABORATORIES, INC. MM-69-8232-1

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES -HO

OFFICIAL FILE COPY
(FORM E-7770) - PLUS
ONE WHITE COPY FOR
EACH ADDITIONAL
FILING CASE
REFERENCED

DATE FILE COPY
(FORM E-1328)

13 REFERENCE COPIES

PATENT DIVISION
IF MEMORANDUM HAS
PATENT SIGNIFICANCE

A232 SUP
CIGP
CIMM

ALLEN,R C
ALTERMAN,M E
ANDERSON,R R
ARNETT,R F
ARTHURS,E
ATANASOWSKY,MISS K A
AWATE,P G
BARTLETT,W S
BASFORD,R L
BAYLISS,E T
BIREN,MRS I B
BIXLER,M A
BLINN,J C
BOYCE,W M
BRINSFIELD,MRS J G
BRINSFIELD,W A
BROWN,P F JR
BUDIANSKY,MRS J
*BURKE,D R
*BUSCH,K J
CANADAY,R H
CAPRAN,J H
CARROLL,J J
CELESTINO,MRS S
CHAPPHA,P L
CHAI,D T
CHANG,S C
CHERNAK,J
CHICK,A J
CHRISTENSEN,C
CHU,W
*CLIFFORD,R M
CONNERS,R R
COOK,J S
CROWLEY,T H
DAUGHERTY,T H
DAVIS,R L JR
DEEM,G S
DEININGER,R L
DENNEDY,MISS E M
DEUTSCH,D N
DOLORDWSKY,I
DORNELL,R T
DORSETT,J R
DRAKE,MRS L
DUFFY,F P
EPSTEIN,M P
FELTON,W A
FINK,MRS B A
FISHER,G M C
FLEISCHER,M I
FLYNN,MISS M L
FOWLKES,E R
FRANK,MISS A J
FREENY,MRS A E
FRETWELL,L J
*FREY,H C
FROST,H B
FUCHS,E
GARRF,J D
GEPNER,J R
GOLDSTEIN,A C
GOLDSTEIN,A J

COMPLETE MEMORANDUM TO

GOODELL,R C
GRAHAM,MRS N L
GREENFIELD,M
GREENWOOD,T S
HALE,A L
HALL,W G
HAMILTON,MISS P A
HAMMER,MRS M T
HARASYMIW,MISS J
*HARMON,L D
HARRINGTON,R J
HASZTO,E D
HAUSE,A D
HAWKINS,R B
HAYNIE,G D
HAZINSKI,J J
HERSHEY,H A
HORZEPA,J J
HUMCKE,D J
HUPKA,MRS F
IWERKS,W C
JACKSON,E K
JACOBS,I
JARVIS,J F
JENSEN,O C
JOHNSON,S C
JOHNSON,W C
JULESZ,B
KAENEL,R A
KAISER,J F
KALISH,M M
KARAFIN,B J
KASPER,F J
KASSON,R A
KELLER,MISS K M
KELLY,L J
KERN,W F JR
KETTLER,M W
KOPEL,P S
KORNEGAY,R L
KROPFL,W J
LANG,J J
LEE,W C
LEWART,C R
LIAUGAUDAS,A P
LOYD,D G
[illegible],G W R
MACLENNAN,MISS C G
MANSELL,J
MC EOWEN,J R
MC TIGUE,G F
MCDONALD,M S
MERMELSTEIN,P
MILLER,J A
MILNE,D C
MOHL,G A
MORGAN,S P
MORRIS,A A JR
MURRAY,R R
MYERS,F H
*NEVILLE,R M
NIMTZ,R O
NINKE,W H
NOLL,MRS L W
OBERER,E
OLLMAN,R T
ORTEL,W C G
OSSANNA,J F JR
PAN,J W
PARDOVICH,J T
PATTON,G C
PETERS,L
PFAFFLIN,MRS S M
PHILLIPS,S J
PHIPPS,G M
PILLA,M A
PINSON,E N
*PLOTT,R S
POAGE,J F
POSTON,W A
PRINTZ,G W
PUERLING,R W
RASPANTI,M
RAWSON,E G
REDLICH,W
REICHERT,R S

COMPLETE MEMORANDUM TO

REPSHER,W G
RITCHIE,D M
ROBBINS,MRS M F
ROBERTS,A W
ROODY,R J
RODRIGUEZ,F J
ROSENTHAL,C W
ROSLER,L
RUSEK,R W
RYAN,W
SALMON,MISS R L
SATZ,L R
SCHAFFER,W G
SCHEINMAN,A H
SCHONBECK,R G
SEVERINGHAUS,J T
[illegible]
[illegible]
SHERRILL,D A
SILAGGI,R C
SINDEN,F W
SINDWITZ,N R
SITAR,F J
SKUBIAK,M T
SMITH,L T
SMITH,M T JR
SNYDER,F W
SNYDER,L C
SO,H C
SPANG,T G
STONEBACK,J P
STURALFFIELD,R W
STUBAS,C D
STUKAS,MRS L I
SWANSON,J F
TANN,MISS P A
THIEMER,R F
THOMPSON,J S
THOMPSON,R L
TUKEY,J W
TUKEY,P A
VAN HAUSEN,J D
VARTABEDIAN,A
VOYTEK,F J JR
WAGNER,MRS M R
WALKER,MISS F A
WANSHAW,A M
WATSON,D S
WEBER,J H
WEXELBLAT,R L
WHIPPLE,J M
WHITE,F W
WILK,M B
WILLIAMS,W H
YAMIN,M
YAMIN,MRS E E
YEISLEY,P A
207 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES -HO

5 COPIES
PLUS ONE COPY FOR
EACH ADDITIONAL
FILING SUBJECT

ABRAHAM,S A
AHO,A V
AHRENS,R B
ANDERSON,T F
APEN,J R
ATAL,B S
ATKIN,R S
BAIRD,D R
BALDWIN,G L
BARGE,R F
BASFORD,MRS N L
BEHLER,MISS B
BERKLEY,D A
BILINSKI,D J

COVER SHEET ONLY TO

BILOWOS,R M
BOHLING,MISS D M
BRAZINSKAS,MRS E E
BRELSFORD,W M
BRICKER,P D
BRISSON,R J
BROWN,W STANLEY
BUSINGER,P A
BUTLER,C M
CALESSO,G L
CARROLL,J DOUGLAS
CASPERS,MRS R E
CEMASHKO,F
CHAMBERS,J M
COCHRAN,J ALAN
COKE,MISS E U
COMELLA,M K
CUTLER,V H JR
DE CHAINE,T L
DIMMICK,J O
DOLE,P F
DOMPIERRE,J A
EHRLICH,N
ESSERMAN,A R
FABISCH,M P
FELDMAN,P M
FELS,A M
FITZSIMONS,T F
FOCARILE,J P
FORD,G A
FRASER,A G
FRIEDES,A
FRIEDMAN,A J
FROEHLICH,F E
GIGR,K R
GITHENS,J A
GITTEN,L J
GOODKIN,N M
GOODMAN,M L
GREENE,MRS D A
GREENHALGH,H W
GRIFFITH,R J
GUST,V J
HAIG,R D
HALLINE,F G
HALL,A D
HARTWELL,W T
HERGENHAN,C R
HERRIOTT,D R
HESSE,MISS D I
HONIG,W L
IVIE,F L
JAKIELSKI,C F
JENKINS,MRS J L
JOHNSTONE,P N
KANKOWSKI,F F
KAPPEL,J G
KARP,S S
KARSNER,MISS C P
KENNEDY,R A
KERNIGHAN,B W
KESSLER,JAMES E
KNOLL,R L
KOLERS,P A
KRANZMANN,R F
KRASNAKEVICH,J R
KRIEWALL,T J
KRUSKAL,J B
LA VITA,MRS P P
LACAVA,MRS B R
LAMBERT,MRS C A
LAMBERT,P F
LANZEROTTI,L J
LAWSON,D A
LEHDER,MRS D M
LEHDER,W E JR
LEVINE,R
LEVITT,H
LIND,J O
LONG,T R
MAC WILLIAMS,W H JR
MAGNANI,R
MALCOLM,J A
MAMMEL,MRS W L
MC CULLOIGH,R H
MC DONALD,R A

COVER SHEET ONLY TO

MC LAUGHLIN,C D
MC MAHIN,L E
MELVILLE,J R
MENIST,D B
MEYER,J S
MILLER,MISS C G
MILLMISTER,W A
MILLS,MISS A D
MOORE,F R
MORGANSTEIN,S J
MORRIS,F E
MORROW,J P
MUELLER,R G
MUENCH,P E
MUGGLIN,M G
MURRAY,R A
MUSA,J D
NASH,D H
NAUGHTON,J J JR
NEHRLICH,W R
NELSON,W L
NEUMANN,P G
NEVERGOLD,R U
NEWHALL,E F
OPFERMAN,D C
ORR,W H
OWENS,F I
PARDEE,S
PETERSON,R W
POLLAK,H O
POLONSKY,I P
PRICE,MRS R
PRUZANSKY,MISS S
RABINER,L R
READY,F A III
RECKER,N J
REED,S C
REHERT,A F
REUTELHUBER,W O
RICHMAN,P L
RIDGWAY,W C III
RIESZ,G W
RINALDY,F A
RISSET,J C
ROBERTS,C S
ROSSON,J W
ROSS,M R
ROTHROCK,H I JR
ROTH,R D
RUSSELL,L K
SCHAEVITZ,A Y
SCHLICHETTI,L L
SCHUPTER,W H
SFOLMAYR,F R
SEIDMAN,L P
SETHI,R
SHANAHAN,R L
SHAO,T S
SIMMONS,D R
SIMMS,R L JR
SIPFLE,W K
SOLOWAY,B H
SPAHN,M J
SPERLING,G
SPILOTRAS,A
STILLERMAN,R
STRITTER,E P
STURGF,M D
TAGUE,B A
TENDICK,F H JR
THOMPSON,M J
THOMSON,MRS M L
TIDWELL,O
TOMAMICHEL,MISS K A
TRAUB,J F
TROTTER,E T
VAN ROOSBROECK,W W
WASSERMAN,MRS Z
WATTERS,R J III
WELLER,D R
WELLS,D J
WESTERMAN,M R
WHITE,J P
WIGGS,J W
WILKENS,E J JR
WILKINSON,F J

* NAMED BY AUTHOR > CITED AS REFERENCE SOURCE

BELL TELEPHONE LABORATORIES, INC. MM-69-8232-1

DISTRIBUTION CONTINUED
(REFER GEI 13.9-3)

COVER SHEET ONLY TO

WINCKEL MANN,W A
WOLFF,R M
WOLPE,G L
YOSTPILLE,J J
ZISLIS,P M
ZWEIG,R
192 NAMES

Power to the computers: A revolution in history?

by SHELDON HACKNEY

Princeton University
Princeton, New Jersey

The first stage in the process of modernizing the historian's intellectual technology is over. No longer is the True Believer's claim to methodological superiority through the computer met by the Luddite's petulent insistence that the most important questions are important precisely because they cannot be quantified.[1] Like the fountain pen and the typewriter before it, the computer is now accepted as a tool that can make a historian's life more pleasant and more productive. The question to be resolved in the next phase of the process is whether the forces activated by the increasing use of computers will or should, either through a sociological dislocation in the profession or a methodological reorientation of the craft, work a revolution in the way history is written. Will the New History be a social science?

The issue has not been squarely faced as yet. When C. Vann Woodward, one of the more genial of the skeptical humanists, recently sought to calm and rally the counterrevolutionary forces he pointed reassuringly to the common humanistic origins of History and the social sciences and suggested that in order to defend this heritage "a small cadre should definitely be armed with all the weapons, trained in all the techniques, and schooled in the ideology of the invaders."[2] This is not the rhetoric of reconciliation, but neither is it a call for a *Kulturkampf*. It is instead an assertion of the belief that because History is not one of the social sciences, historians may borrow from them without capitulating to them, a belief shared by Woodward and an impressive cohort of practicing historians: R. R. Palmer, J. H. Hexter, David M. Potter, Richard Hofstadter and H. Stuart Hughes among others. One suspects that the premise of this confidently resilient response is that "far from being revolutionized by new techniques, transformed beyond recognition, or swallowed up by the social sciences, much the greater part of history as written in the United States has remained obstinately, almost imperviously traditional."[3]

The premise is valid. Computers and the methodological strain toward quantification have not yet altered the way historians go about their work. In the first place, the historians were consciously quantifying even without computers. The classical quantitative study of violence during the French Revolution was done in 1935,[4] and one of the most impressive pieces of recent scholarship, The *Crisis of the Aristocracy* by Lawrence Stone, uses a host of noncomputerized numerical measurements of social behavior to demonstrate the existence of a crisis in the affairs of the elite that led to the fact that the English aristocracy in the early 17th century experienced a marked decline in prestige and deference. The author infers from this that the collapse of the authority of the peerage underlies the coming of the English Civil War.[5] Large, old questions can be answered by numbers even without the aid of computers.

This is not to argue that computers are unnecessary. The information contained in the Massachusetts ship registry for the years 1697–1714 remained locked there until the computer and a willing programmer combined to make it available and easily manipulable in tabular form. Then simple but important questions could be answered, and historians now may state with greater confidence that even at the end of the seventeenth century the Massachusetts shipping industry was geographically dispersed and socially diffused.[6] The myth of American opportunity had some basis in fact.

Examples of the usefulness of computers may be drawn from all of the areas into which computers have made incursions. One of the most promising yet underexploited fields is historical demography, in which the computer is a near necessity because massive amounts of data must be manipulated in order to get significant answers. The most sophisticated work has been done in European history without the aid of computers, but Herbert Guttman and Laurence A. Glasco mechanized census data for their forthcoming study of the related

quest'on of black family structure in nineteenth century American cities and found a great deal more stability than Daniel Moynihan would have predicted.[7] In the similarly related and highly developed area of social mobility studies, Stephan Thernstrom used computers to process census data for Newburyport, Massachusetts, over a period in the mid-nineteenth century and concluded that even though there was not enough occupational mobility to justify the rags-to-riches myth, there was an impressive amount of upward mobility of blue-collar families if property accumulation were used as the criterion.[8] The interesting thing about this example of computer employment, for present purposes, is that Thernstrom was using familiar sorts of data in a more rigorous way to find answers to questions historians had been dealing with for some time.

Historians have realized for a long time that the study of political behavior demanded quantitative techniques. As early as 1896 Orin Libby called upon American historians for close analysis of Congressional voting behavior, and he set an early example in the area of electoral behavior with his study of the geographical distribution of the vote on the ratification of the Constitution in 1787–88.[9] The age of computers has stimulated a leap in the quantity of quantification but only a small increment so far in the level of sophistication of the analyses.[10] Thomas Alexander and his associates used simple correlation analysis of beat returns with measures of socio-economic status in Alabama to construct the most convincing argument so far that the American Whigs in the 1830s and 1840s were not a class party.[11] Stanley Parsons employed slightly more advanced multiple correlation techniques on similar data to demonstrate that there is little substance to the conventional widsom that Populism and mortgage indebtedness in the 1890s in Nebraska went hand-in-hand.[12] Intercorrelations of election returns over a long period of time revealed that critics who found the source of Joe McCarthyism in mass democracy were incorrect in supposing that McCarthy's support in Wisconsin came from the same elements of the population that had supported Populism in the 1890s.[13] Sheldon Hackney resorted to cluster-bloc analysis to identify four different groups representing different sets of political values in the Alabama Constitutional Convention of 1901 and then used these as categories around which to organize subsequent political developments in the state.[14] Guttman scaling is another popular method being used by political historians. With it, Joel Silbey was able to demonstrate that sectional conflict in the 1840s and 1850s did not replace national party rivalry as the principal dimension of Congressional politics.[15] With the accumulation of data archives and standard programs by the Inter-University Consortium for Political Research at Ann Arbor and the increasing mathematical competence among historians being stimulated by the Mathematics Social Science Board, there could be a leap in both the quantity and quality of political history in the near future.

Social behavior other than in the realm of politics is also being quantified and analyzed profitably with the aid of computers. Multiple correlation analysis using a dummy variable, applied to social and economic variables and homicide and suicide statistics, has disclosed that there is a non-quantifiable cultural component associated with the high rates of individual violence in the American South so that regional differentials cannot be explained totally by differences in rurality, poverty, and generally lower levels of modernization.[16] Charles Tilly, a sociologist who uses history as his laboratory, is altering conventional assumptions about the process of urbanization through a massive study of collective violence in France in the nineteenth and twentieth centuries.[17] Similarly, Michael Katz has used factor analysis and other techniques with a wide assortment of data to establish a positive correlation between urban growth and educational reform in nineteenth century Massachusetts, and has developed a set of hypotheses consistent with this finding. Educational reform, he argues, was not undertaken in response to the pressure of upward aspiring lower orders; it was the result of a coalition of upper status groups each pursuing slightly different but temporarily compatible goals so that the creation of high schools can best be considered as a reform sponsored by elite groups to provide social control in a situation of rapid modernization from which they were profiting.

Collective biography offers another approach to historical problems that historians are just beginning to systematically exploit with the aid of the computer and the creation of a central data bank at Princeton University. The leader in this endeavor is Theodore K. Rabb whose extensive analysis of 5,184 investors in seventeenth century English trading ventures firmly established the fact that the landed gentry in England supported imperial enterprise and industry to a unique extent.[19] Ralph Wooseter's massive analysis of the membership of secession conventions in the southern states in 1860–61 confirms the notion that slaveholders were in control of at least this stage of the secession movement.[20] Collective biographies need not always confirm existing interpretations, however. When Norman Wilensky constructed social profiles of a large sample of Republican Party activists in 1912, he discovered that stand-pat Republicans did not differ significantly from progressive Republicans in any social

characteristic other than age, and this cast great doubt on the Mowry–Hofstadter notion that one of the main sources of progressive leadership was downwardly mobile elites.[21] Even so, it is apparent that, thus far, computer-aided quantified biographical studies are testing hypotheses and answering questions long familiar to more traditional methods of historical scholarship. Computerized collective biography represents only an extension of these more traditional techniques.

The field of History that comes closest to having undergone the sort of transformation that the study of economics experienced with the development of econometrics is, quite naturally, economic history.[22] In this area, as in the others, the computer is not the only force making for changes in technique, for there is also a marked adventurousness in the use of concepts and models borrowed from the social sciences and in the conscious construction and testing of hypotheses. Such theorizing in History as in pure math or theoretical physics does not depend on the availability of powerful computers. Even without computers we would still today find traditional modes of economic history, such as business and labor history, being surpassed in importance by quantitative analyses of economic systems based on highly articulated theoretical constructs. The two best examples of the new economic history are the debate over the profitability of slavery rekindled by Alfred Conrad and John Meyer[23] and the argument between Albert Fishlow and Robert Fogel about the importance of railroads to American economic growth in the nineteenth century.[24] But both of these cases, as important as they are, represent attempts at more precise measurements and theoretical assumptions involved in answering big questions that historians had been dealing with for years. Even in the most advanced branch of cliometrics, the revolution is incomplete.

Because of the arrested state of change, all but the grumpiest humanists tend now to overlook the methodological peculiarities of the computerized upstarts and to content themselves with the thought that history is not a social science. They argue at times that historians are not interested in formulating general laws, Arnold Toynbee to the contrary notwithstanding. They insist that historians are not concerned with the regularities of human behavior but with particular and unique events. Some humanists would even like to establish the axiom that social scientists are interested in discovering general rules of human behavior and historians are interested in the exceptions. But on examination this turns out to be a faulty argument. Only the antiquarian is fascinated with the artifacts, events, modes of life, and personalities of the past for their own sake. Historians seek to establish interrelationships, and interrelationships imply causal connections. Even when historians are not consciously doing social science in the sense of making and testing hypotheses concerning the relationship of two or more variables, they are dealing with such hypotheses, usually as unconscious assumptions or rejected explanations. Imagine how a historian's explanation of an event would fare if the causal conditions he points to appeared nowhere else in human experience in conjunction with events similar to the one he is attempting to explain. All history is at least implicitly comparative, and what is comparison if it is not hypothesis testing?

A fundamentally more sound objection is the conviction of many historians that more forces are at work in a given situation than can possibly be reconstructed and abstracted by the scholar. Even though we need to explain only the most important of the causes, and not all of them, these scholars believe that truth may be more closely approached by the narration of the story, reflecting the historicist belief that stopping the flow of history does violence to understanding. Historians share with humanists the habit of leaving much of the job of understanding to the reader and providing him with a superfluity of facts which he can fit into his own scheme. The humanities depend to a great extent on a shared culture, and books written for one audience are not necessarily understood by another audience. For instance, the hero of William Styron's novel, *The Confessions of Nat Turner*, was intended by the author and is understood by most white readers to be an existential hero striking out against oppression in the face of incredible odds. Black readers, because of their different definition of the attributes of manhood, interpret Styron's Nat Turner as an insult to the race because he is made to appear weak, ineffectual, indecisive, and neurotic. The only way to close the gap in understanding is to discuss why the fictional Nat Turner acted as he did in certain situations and whether this behavior is consistent with what we know of Turner's biography, the situation, and human behavior in general. Understanding, for the historian as for the social scientist, is finding the answers to a set of "why" questions.

There is no essential sense in which History differs from the social sciences.[25] Though historians still think of themselves as belonging to a literary craft, they do not usually confuse beauty and truth as Lord Byron did. For this reason, though no great change has yet occurred in the kinds of questions historians ask, one must not assume that the revolution will never come.[26] The pressure toward further change is evident in the *Historical Methods Newsletter* whose focus on quantitative methodology testifies to a new orientation and

whose book reviews frequently, and correctly, charge authors with not squeezing enough out of the analytical tools available to them. Under the new orientation, the historian should not stop his analysis after he has answered the question that led him to adopt the quantitative technique, but he should push on until the possibilities of the technique and the data have been exhausted. Depending on one's point of view, this approach produces either unwanted knowledge or a strikingly new level of analysis. So, the principal dichotomy is no longer between social science oriented revolutionaries and humanistic traditionalists, but between those historians who view the computer as a tool to be used only to the extent that it is useful in answering previously determined questions and those who advocate total immersion in computer and quantitative techniques in hopes that completely different kinds of questions will eventually be posed and answered.

Now is the time to stop and think carefully about the likely consequences should the total immersionists triumph. Is there something about the study of history that would be destroyed by applying the research strategies of the social sciences? There may well be.

Even though historians and social scientists strive for the same kind of understanding, historians retain from their humanistic past certain work habits that are important. Historians are usually specialists in particular times and places, not in problems or techniques. Even specialists in urban, economic, social, or political history tend to focus their interests on particular geographic locations and chronological periods. They attempt to understand the particular event in relation to the general rules, rather than trying to derive the general from the particular, and they have an imprecise faith that, as Pirenne said, "to construct history is to narrate it." It may be an absurd myth to believe that historical truth depends upon a holistic approach, or that the whole is greater than the sum of its parts, but it is a useful myth. Because historians believe that truth emerges from the complete context, they seek to synthesize all of the pieces. Because they proceed as they do, they are likely to discover that some previously unsuspected factors are pertinent to a particular problem, and they are more likely to provide information useful to a future scholar with a different set of concerns.

This picture of the myth of historical truth is reflected in the present caste structure of the profession. The generalists rule and the technicians execute. How a cliometrical revolution would affect this structure is an interesting question. Generalists probably would still dominate the profession, but there would be a difference. Only those synthesizers well versed enough in the new techniques to be able to understand and judge the reliability of the work of the technical specialists could survive. That would be a clear gain. But what if it became true that the only historical scholarship admired and respected were computer-aided quantification because that is the only sort of scholarship that provides neat answers to generalized hypotheses. That would be a tragedy, for we would lose the habit of relating different areas of human endeavor to each other, the habit of syntactical analysis.

If historians can learn from the history of other disciplines that have undergone methodological revolutions, they will leapfrog over the next stage in their revolution, the stage during which so much attention is paid to methodology that little productive energy is expended in advancing the understanding of the substance of history. No contribution to any field can be made by scholars not steeped in the substance of it. Even some of the most methodologically oriented of the new breed are discovering the usefulness of traditional methods. As Robert Zemsky has recently warned, quantitative techniques can yield answers no better than the measurements on which they rest and the only way for the quantifier to avoid misuse of the measurements is to understand the record through traditional approaches.[27]

We need to arrive quickly at that point in the development of the discipline at which methodology is no longer the central issue. Then, techniques will neither be damned because they are new nor pursued for their own sake. New techniques are more likely to be generated by new questions than new questions are to be created as a byproduct of new techniques. It would be a mistake for traditionalists to assume that there is such a great difference between social science and History that there is a limit to the change that can be wrought by the quantifiers, and it would be an even greater error for the quantifiers to assume that there would be little lost should they accomplish their revolution in the work habits of historians.

REFERENCES

1 Examples of the old style rhetoric of conversion can be found in:
L BENSON
Quantification, scientific history, and scholorly innovation
AHS NEWSLETTER June 1966
S P HAYES
The social analysis of American political history, 1880–1920
Political Science Quarterly September 1965
On the Luddite side sample:
C BRIDENBAUGH
The great mutation
American Historical Review January 1963

A M SCHLESINGER JR
The humanist looks at empirical social research
American Sociological Review December 1962
For a sane statement by a leading quantifier:
W O AYDELOTTE
Quantification in history
American Historical Review April 1966

2 *History and the third culture*
Journal of Contemporary History April 1968

3 Ibid 24

4 D GREER
The incidence of terror during the French revolution: A statistical interpretation
Cambridge Masachusetts 1935

5 *The crisis of the aristocracy 1558–1641*
Oxford 1965

6 B BAILYN L BAILYN
Massachusetts shipping 1697–1714: A statistical study
Cambridge Massachusetts 1959

7 A preview of their findings was contained in a paper delivered at the Yale Conference on the Nineteenth Century Industrial City in November 1968

8 S A THERNSTROM
Poverty and progress: Social mobility in a nineteenth century city
Cambridge Massachusetts 1964

9 O G LIBBY
A plea for the study of votes in Congress
American Historical Association Report I 1896

10 A G BOGUE
United States: The "new" political history
Journal of Contemporary History January 1968

11 T B ALEXANDER K C CARTER J R LISTER
J C OLDSHUE W G SANDLIN
Who were the Alabama whigs?
The Alabama Review January 1963
T B ALEXANDER P J DUCKWORTH
Alabama black belt whigs during secession: A new viewpoint
Ibid July 1964

12 S PARSONS
Who were the Nebraska populists?
Nebraska History June 1963

13 M P ROGIN
The intellectuals and McCarthy: The radical specter
Cambridge Massachusetts 1967

14 *Population to progressivism in Alabama*
Princeton New Jersey 1969

15 *The shrine of party: Congressional voting behavior 1841–1852*
Pittsburgh 1967

16 S HACKNEY
Southern violence
American Historical Review February 1969

17 C TILLY
Collective violence in European perspective
Violence in America: Historical and Comparative Perspectives (eds H D Graham and T R Gurr) New York 1969

18 M KATZ
The irony of early school reform: Educational innovation in mid-nineteenth century Massachusetts
Cambridge Massachusetts 1968

19 T K RABB
Enterprise and empire: Merchant and gentry investment in the expansion of England 1575–1630
Cambridge Massachusetts 1967

20 R WOOSETER
The secession conventions of the south
Princeton 1962

21 N M WILENSKY
Conservatives in the progressive era: The Taft republicans of 1912
Gainesvile Florida 1965

22 R W FOGEL
The new economic history, its findings and methods
Economic History Review December 1969
T C COCHRAN
Economic history, old and new
American Historical Review June 1969

23 CONRAD MEYER
The economics of slavery and other studies in econometric history
Chicago 1964

24 R W FOGEL
Railroads and American economic growth: Essays in econometric history
Baltimore 1964
A FISHLOW
American railroads and the transformation of the ante-bellum economy
Cambridge Massachusetts 1965

25 A reliable statement of this position is found in Chapter 15 of:
E NAGEL
The structure of science: Problems in the logic of scientific explanation
New York 1961

26 The differences between the new and the old history is minimized in the introduction of their excellent collection of essays in:
D K ROWNEY J Q GRAHAM
Quantitative history: Selected readings in the quantitative analysis of historical data
Homewood Illinois 1969
A more confidently revolutionary view is vocied by:
J M CLUBB H W ALLEN
Computers and historical studies
The Journal of American History December 1967

27 R M ZEMSKY
Numbers and history: The dilemma of measurement
Computers and the Humanities September 1969

Music and the computer in the sixties

by RAYMOND F. ERICKSON

Yale University
New Haven, Connecticut

INTRODUCTION

A glance at the cumulative bibliographies published in *Computers and the Humanities* each year provides ample evidence that there is a striking amount of interest in computer applications for both the art of music and the discipline of musicology. Allen Forte, reporting to the 1967 Fall Joint Computer Conference,[1] noted that connections between music and computing are by no means unique to our time, but extend all the way back into antiquity, when the mathematical ratios of musical intervals were believed to mirror and explain the order of the universe. He then outlined the areas within music using computers and summarized the types of application involved, stressing, however, that much of the work being done was experimental: "It is necessary to say, once and for all, that we are still in a pioneer stage".

The writer of this summary of the situation at the close of the first decade of computer use in musical studies proposes to update Forte's observations of three years ago, reserving until the end an evaluation of the progress we have made.

ACOUSTICS, SOUND SYNTHESIS AND COMPUTER COMPOSITION

As a means of sound analysis and generation, the computer has had far-reaching applications and implications. On the one hand, it has provided a common tool for the physicist, composer and musical scholar interested in the nature of "musical" sound and of the acoustical properties of musical instruments. Closely allied, also, is the field of psychoacoustics. Thus, a perusal of the *Journal of the Acoustical Society of America* as well as certain psychology and general scientific periodicals can yield insights useful to the researcher or composer using computers.

Narrowing our view to the specific topic of computer music, one finds that many of composers mentioned by Forte have since documented their work more fully. Max Mathews has devoted an entire book to *The Technology of Computer Music*,[2] which includes considerable information on the use of his MUSIC V program for sound synthesis. (The MUSIC series of programs, developed primarily at Bell Labs, has become the most widely used means for synthesized sound generation.) Lejaren Hiller, now at SUNY (Buffalo), is another of the older generation of computer composers; his work is particularly notable for the degree in which the computer is used to generate ("compose", if you will) the material to be synthesized.[3] Exception to the Hiller approach has been taken by A. Wayne Slawson (Yale),[4] whose own compositions are distinguished by experimentation with the musical possibilities of the sounds of speech.[5] Slawson, who holds a doctorate in psychology, exemplifies the composer whose work has been influenced by acquaintance with research going on in related areas of acoustics and psychology.

Turning now to the vast territory of computer-assisted research, it might be well to begin by checking up on the current state of projects specifically mentioned in the 1967 paper.

BIBLIOGRAPHIES AND CONCORDANCES

Perhaps the most ambitious and important undertaking in the area of musical studies is RILM[6] (**R**epertoire **I**nternational de la **L**itterature **M**usicale), a computer-indexed bibliographic publication covering the scholarly literature in music and containing abstracts as well as the normal cross-indexed entries. Happily, the first issues of RILM have appeared since 1967, although the current number is overdue. Let us fervently hope that RILM is one publication that will survive, for it is badly needed.

Word from Professor Arthur Mendel of Princeton University just received states that a concordance of

Bach vocal texts announced earlier is virtually complete. This concordance could prove valuable to the student or scholar studying the relationships of certain words and their musical settings in the Bach cantatas.

Prof. Harry Lincoln (SUNY: Binghamton) has been indefatigable in his attempt to encode and catalog melodic incipits of certain categories of Italian Renaissance music.[7] Since Forte's report, the thematic index of the *frottola* repertory has been completed along with incipits of virtually all of the music of Palestrina. Now attention is being turned to the Italian madrigal, 1530–50. Professor Lincoln hopes to make Binghamton a central clearing-house for data bases of this type.

Other notable thematic catalogs now being compiled for computer processing are of the 18th-century symphony[8] (Jan LaRue *et al* at NYU), the complete works of Mozart[9] (George B. Hill, Jan LaRue *et al* at NYU), and materials by and related to Bartok[10] (Benjamin Suchoff).

STYLISTIC ANALYSIS

No matter how niftily they may be encoded, thematic incipits can do little or nothing for the theorist or historian interested in subtle problems of compositional technique and in discovering any but the most basic interrelationships among compositions. However, even though the area of stylistic analysis offers the most enticing opportunities for harnessing the full power of the computer, it requires a fairly sophisticated methodology and considerable programming support. These factors have prevented many from attempting anything more than the most trivial type of musical analysis and thus it is not surprising that three years ago there was little activity to describe.

Of particular interest, then, was the recently announced decision of Prof. Lawrence Bernstein (University of Chicago) to abandon the thematic index of the 16th-century chanson (begun in 1964 and encompassing ⅔ of the 6000 extant chansons) in favor of complete encoding of the chanson compositions. His reason:

> Inasmuch as the thematic index identifies a composition according to the opening musical material of each piece, it cannot, perforce, discriminate accurately in two circumstances: (1) when pieces that are essentially the same begin with different material; and, conversely (2) when pieces that are essentially different begin with the same material. . . . A third problem gives reason for further concern. Composers who based their chansons on preexistent pieces may have borrowed musical material from internal points within their models. An ideal index of the chanson repertory would reveal such relationships, but one based on musical incipits alone would not, of course, be able to do so.[11]

The data sets resulting from the new procedure will make possible not only a thematic concordance, but will open the doors to a comprehensive computer-assisted stylistic study of the repertory that could not have been undertaken before.

Those not conversant with the issues that have raised the most hackles in discussions regarding computer-aided musical analysis may not fully appreciate the significance of this change of tactics; however, a debate has been raging for years over the questions of how much information one should encode and how one should do it. One partisan group, headed by Stefan Bauer–Mengelberg and Allen Forte, has advocated from the start the representation of entire scores if an investigation might even remotely involve stylistic analysis. It would not, of course, be difficult to extract incipits from musical data thus encoded.

Because the nature and scope of various projects differ, there was an initial tendency for each researcher to develop his own encoding procedures which, once implemented, were defended with tenacity. Of all the various systems, however, only one was originally conceived with the object of including *all* information found in a printed score—for the simple reason that it was to be used with a computer-driven music printer. This is the Ford–Columbia Music Representation (DARMS) designed by Stefan Bauer–Mengelberg, which has now emerged as the most widely used encoding system (and, incidentally, is the model for Prof. Bernstein's current approach). It stands as the most likely candidate for an encoding standard, the *sine qua non* for really significant progress in computer-aided musical research.

Languages like DARMS (or Plaine and Easie Code[12] or ALMA,[13] two of its competitors) are relatively easy to use but can result in free-form data strings of great length and complexity. Inevitably this complicates the necessary and tedious process of checking for encoding and keypunching errors. It has also caused many to take the easy road of partial encoding, i.e., leaving out categories of information deemed of little or no relevance. But, as Prof. Bernstein points out

> . . . once a complete piece of music is translated into a machine-readable language (ostensibly so that it might be included in a thematic concordance), peripheral benefits begin rapidly to accrue. And some of these advantages, we felt, outweigh in usefulness the original purpose for which the music had been encoded.[14]

What will be relevant, then, is not always obvious. But to encourage rigorous and comprehensive encoding procedures, some sort of syntax analyzer is necessary to speed up the clerical tasks of error detection and correction. Facilities of this type were built into the independent systems for musical analysis designed and implemented by Eric Regener (SAM)[15] and Michael Kassler and Tobias Robison (IML-MIR),[16] all at Princeton when the projects were begun. More recently Eors N. Ferentzy (University of Toronto),[17] George Logemann (NYU) and the present writer (Yale)[18] have been working on syntax-directed compilers for processing musical data for syntax and stylistic analysis.

Among the investigations of specific musical repertories using computer-aided stylistic analysis may be cited (in their historical order) those of late 12th-century polyphony (the author), the Masses of Josquin (Arthur Mendel and Lewis Lockwood), the 16th-century chanson (Lawrence Bernstein and Joseph P. Olive), the 18th-century symphony (Jan LaRue and Murray Gould), Anton Webern (Mary E. Fiore), and non-tonal music of the twentieth century (Allen Forte).

There is also research in musical graphics being pursued by Jeffrey Raskin (University of California, San Diego) and Barry Brook and Richard Golden (Queens College). This would probably also be the place to mention the availability of a special print chain with the symbols of musical notation at SUNY (Binghamton).

EVALUATION

From the above account it should be clear that not only are computers being employed in a wide variety of ways in music, but that this activity is no longer concentrated at a few centers. But, examining the record more closely, how much has actually been accomplished in terms of concrete research results or completed projects?

It is with some embarrassment that I report that there is little to report.

Virtually all documented research dealing with music itself (which is, we sometimes forget, our main concern) has been of limited scope (Fiore,[19] Bernstein,[20] Mendel—see below, and Forte[21]) but even more sobering is the number of projects abandoned or inactive. Since the Forte account of 1967 we have seen the demise of the *Electronic Music Review* and the Institute for Computer Research in the Humanities at NYU, the cessation of work on the photon music printer for which DARMS was originally intended, and the obsolescence of the Princeton systems in which so much hope had been placed. All of these setbacks are of a high order.

Instructive in content and praiseworthy for its frankness is a recent article by Arthur Mendel which, in effect, announced the termination of (or at least a significant hiatus in) the progress of one of the first (1963) and most publicized undertakings in style analysis: the so-called "Josquin project" mentioned above. The music, now completely encoded in Kassler's IML was to be analyzed by programs written in the special-purpose programming language (also designed by Kassler) known as MIR. However, the MIR compiler was written in BEFAP for the 7094 and since "the debugging process was completed about March 25 of this year, and on April 1 the 7094 for which the system was designed left Princeton,"[22] the data is of little use at present.

The obvious lesson to be learned here is that it simply doesn't pay, given the flexibility of high-level programming languages such as PL/I and SNOBOL, to program at the assembly language level for humanities applications. (There is an alarming tendency among humanists who program to become so involved in the computing aspects of their work that the substantive problems become secondary.) I would also add that the recent developments in high-level languages, especially PL/I, obviate the need—quite legitimately felt at the time MIR was conceived—for special-purpose programming languages. On the other hand, the inadequacy of FORTRAN for non-scientific applications should be stressed, despite the claims of some (who write machine dependent FORTRAN, which defeats the purpose) that it can do anything PL/I can do better and faster. The potential of APL for musical studies has not yet been fully evaluated, although Mr. Christian Granger at SUNY (Binghamton), for one, has been experimenting along these lines.

The realization of the discrepancy between the enthusiastic claims of five years ago and the hard facts of production today has gradually been tempering the optimism of humanists. I take this to be a positive sign, for if the current generation of graduate students is informed with a healthy skepticism, we shall probably end up with a core of scholars who are dedicated, sophisticated and rigorous in their methods, and who will have profited from the mistakes of this first decade. The Harpur College summer seminars in Music and Computers have sought to provide this kind of training on a modest scope as have special courses in a few music departments around the country. Furthermore, in September, 1969, a group of 40 experienced scholars gathered with representatives from industry to lay plans for an Institute devoted to the education of non-science students and faculty in relevant computer applications.[23]

There have been other developments that augur well

for the future. The spectacular commercial success of the Moog synthesizer recordings[24] (whether or not one prefers his Bach "Switched-On" is beside the point) has suddenly made the general public aware of the virtually unlimited spectrum of synthesized sounds. This could conceivably produce a wave of serious interest in computer music which is now more or less the province of the connoisseur. The repertoire of electronic compositions is surprisingly large but the musical public has been either oblivious of or antipathetic to its existence.[25] Nonetheless, it has long been felt that electronic music is the music of the future. Perhaps the future has just arrived.

The pressing need for an encoding standard was cited above. The only way this can be achieved, I am convinced, is to provide for a particular encoding language elaborate software support (syntax error detection routines, standard output table generators, etc.) in a more or less machine independent language to permit (indeed, insure) wide and ever-increasing implementation. Fortunately, a canonical version of the Ford–Columbia language is almost finished at this writing and plans have been made to develop such supplementary programs.

Therefore, in spite of an unimpressive record of achievement in the past decade, the "pioneer stage," there is reason to believe that we are on the threshold of important breakthroughs in virtually all major areas of computer application in music and that whoever writes a similar report a few years from now will be able to substantiate this prediction.

REFERENCES

1 A FORTE
Music and computing: The present situation
Proceedings of the Fall Joint Computer Conference pp 327–329 1967

2 M V MATHEWS
The technology of computer music
MIT Press Cambridge Massachusetts 1969

3 L A HILLER
Programming a computer for music composition
Computer Applications in Music Ed G Lefkoff pp 65–88 West Virginia University Library Morgantown West Virginia 1967

4 Review of volume cited in Reference 3
Journal of Music Theory XII 1 pp 105-111 1968

5 W A SLAWSON
A speech-oriented synthesizer of computer music
Journal of Music Theory XIII 1 pp 94–127 1969

6 B S BROOK
Music bibliography and the computer
Computer Applications in Music pp 11–27

7 H B LINCOLN
Some criteria and techniques for developing computerized thematic indices
Elektronische Datenverarbeitung in der Musikwissenschaft Ed H Heckmann pp 57–62 Regensburg Gustave Bosse Verlag 1967

8 J LARUE M W COBIN
The Ruge–Seignelay catalogue: An exercise in automated entries
Elektronische Datenverarbeitung in der Musikwissenschaft pp 41–56

9 G HILL
The thematic index to the Kochel catalog
Institute for Computer Research in the Humanities Newsletter IV 7 8f New York University 1968

10 B SUCHOFF
Bartok, ethnomusicology and the computer
ICRH Newsletter V 4 pp 3–6 1968

11 L F BERNSTEIN J P OLIVE
Computers and the 16th century chanson: A pilot project at the University of Chicago
Computers and the Humanities III 3 pp 153–160 1969

12 Described in the article cited in footnote 6

13 G W LOGEMANN M GOULD
ALMA, Alphanumeric Language for Music Analysis
Institute for Computer Research in the Humanities New York 1966

14 BERNSTEIN
op cit 156

15 E REGENER
A multiple-pass transcription and a system for music analysis by computer
Elektronische Datenverarbeitung pp 89–102 For a critique of this and the article cited in the next footnote see:
R F ERICKSON
Musical analysis and the computer: A report on some current approaches and the outlook for the future
Computers and the Humanities III 2 pp 87–104 1968
This article somewhat expanded was republished in the Journal of Music Theory XII 2 pp 241–263 1968

16 T D ROBISON
IML-MIR: A data-processing system for the analysis of music
Elektronische Datenverarbeitung pp 103–135
See also the Mendel article cited below

17 E N FERENTZY
A syntax director processor writing system
A paper submitted to the IFIP Congress of 1968 (Proceedings unavailable to this writer)

18 R F ERICKSON
A general-purpose system for computer-aided musical studies
Journal of Music Theory XIII 2 1970

19 M E FIORE
Motive in Webern's piano variations
Computer Uses in Music In Press Cornell University Press Ithaca New York

20 See note 11

21 A FORTE
A program for the analytic reading of scores
Journal of Music Theory X pp 330–63 1966—Contains some preliminary results of a study of non-tonal music that is the basis for a forthcoming book

22 A MENDEL
Some preliminary attempts at computer-assisted style analysis in music
Computers and the Humanities IV 1 p 45 1969

23 The Conference on Computer Technology in the Humanities was held at the University of Kansas (Lawrence) September 2–7 1969 and was administered by the ACLS under an NSF grant

Co-chairmen were Professors Earl J. Schweppe (Computer Science) and Floyd R. Horowitz (English) of the host institution

24 *Switched-on Bach* and *The Well-tempered synthesizer* Columbia Records—Robert A Moog was not unknown before these releases however
See his *Introduction to Programmed Control* Electronic Music Review I pp 23–32

25 L M CROSS
A bibliography of electronic music
University of Toronto Press Toronto Canada

Prosody and the computer: A text processor for stylistic analysis

by HERBERT S. DONOW

Southern Illinois University
Carbondale, Illinois

My purpose in this paper is to explain how a computer can be used in prosodic analysis and how the results may be used in stylistic studies. When I began this work nearly two years ago, my aim was to develop a computerized method of classifying lines of poetry with respect to rhythmical features. Since then, I have come to feel that the technique has broader applications, effective for making stylistic discriminations with any kind of language input. An interesting observation about this approach or, perhaps it would be more accurate to say, an interesting observation about language itself is that the linguistic phenomena that I use to tell me things about poetic rhythm are relevant sources of information about prose style as well.

Since the text processing is fairly routine in that it employs techniques in wide use—machine dictionary development, binary search, tagging, and statistical tabulations—I will try to be as brief as possible on these points. The important facet of this paper is the discussion of some assumptions about language that are at the root of my processor. One reservation that I should make is that I do not claim any kind of absolute truth for some of the things that I am about to say. My "cop-out" is that this is a statistical procedure, involving measurement and inference, and not, strictly speaking, linguistic analysis.

Before turning to the substantive issues of this paper, there are a few remarks about style and poetry that I should make. Although we can say that the fundamental objective of language processing, whether we do it by machine or human brain, is to determine meaning, we know that even at basic levels, elements other than content are important. If meaning were all that mattered, a sentence like "Arboreal anthropoids inhabit herbaceous environments" could be indiscriminately substituted for "Tree apes live in forests." It does not tax our imagination or vocabulary to put a finger on the differences between these two sentences, but where the case is less clear, our inadequacies, both in our amorphous conceptions of style and in our lack of a vocabulary, has seriously hampered us. Read most literary critics on style, even today, and you find abundant examples of imprecise and uncommunicative vocabulary at work. For example, Yvor Winters in an essay on "Poetic Style in Shakespeare's Sonnets," says that the first four lines of Sonnet 116 "have precision, dignity, and simplicity, which are moving," and yet if we read those lines,

> Let me not to the marriage of true mindes
> Admit impediments, love is not love
> Which alters when it alteration findes,
> Or bends with the remover to remove.

we would be hard pressed to explain the specifics that led to his judgment. What basis has Winters given his reader to make a distinction between the lines I have just read and almost any other quatrain in the entire corpus?

While subjective reactions are to be welcomed in aesthetic criticism, particularly if the critic impresses us as being trustworthy, I can think of one class of critics who offend egregiously in the aimless subjective characterizations of stylistic phenomena: the critics of student writing. How many compositions have been judged adversely because their styles are "muddy," "turgid," "vague," or "awkward"? Of dubious pedagogical value, such assessments serve only to emphasize our incapacity to communicate perceptions about style.

My own predilections in stylistic analysis is to approach style as a measurable and describable component of language, whether it be belletristic or in ordinary speech, and while I hardly feel up to a total overview of the subject—a comprehensive theory of style—I see a value in working on specific elements. Certainly the work with word frequencies like that done with *The Federalist* papers by Mosteller and Wallace, Milic's studies on Swift, or the work on sentence length in selected Greek authors by Morton and Levison consti-

tute examples of attempts to provide an objective basis for making stylistic distinctions. Part of my enterprise is, therefore, not to say any more about style as a general concept, but about a definable component, for which, momentarily at least, no name exists.

As I said earlier, the work that I have been doing began as a study of poetic rhythm. One reason that I chose to work with poetry rather than stylistic features in prose is that large quantities of English poetry have regular features that make the extremely difficult task of language processing a trifle easier. When I use the word poetry, I am actually referring to a subset that has the following characteristics: (1) a uniform number of syllables per line, preponderantly ten; (2) a pause (or caesura) at the end of each line which signals the end of a syntactic unit (a sentence, clause, phrase, or phrase cluster); (3) a stress pattern which generally finds one unstressed syllable followed by a stressed syllable. In other words, what I am speaking of, in most cases, is endstopped iambic pentameter, or at least endstopped decasyllabic verse.

As anyone with even a nodding acquaintance with poetry can attest, endstopped, iambic pentameter is not always endstopped, not invariably iambic, and in some notable instances, inconsistent in the number of feet in the line. Even the number of syllables occasionally varies from the expected number. But we are talking about the average case; the exceptions do not alter the generalizations in any statistical sense.

The basic premise in the procedure is that information about the frequency of occurrence of polysyllables, the stress patterns, and the ratio of words rich in content value to words with lower levels of content value can permit us to make inferences about (1) the way a line sounds and (2) how one line differs from another. Thus, the poetry is searched for the following features: number of syllables per word, content value, and, in the case of polysyllables, accent placement.

The following assumptions will, I think, explain why words are tagged for these features. First, in sentences composed of a fixed number of syllables, there will be a distinctive difference in rhythmic quality between sentence "a" which averages one syllable per word as opposed to sentence "b" which averages two syllables per word:

a. If John had done as well in school as Bill, he would have passed.
b. Although Richard is uneducated, he has prospered.

Sentence "a" takes almost a second longer for me to recite than sentence "b". Although an analytic examination of the sentences will provide us with some good reasons for this, the fact that the first is composed of fourteen monosyllables while the second contains only seven words is correlated with their phonological difference. Since the number of syllables per word and the number of words per sentence is something that a computer can calculate without great difficulty, we are, at least for our present purposes, more interested in the superficial manifestations than the underlying reasons.

Of course, most of my processing has been with poetry and that precludes the necessity for counting syllables. We already know that a line has ten syllables and, therefore, we need only count the number of words, that is, the number of literal strings between blanks.

I have listed some lines from Shakespeare's sonnets to illustrate what effect the word-count has in a poetic context. Looking at two extremes, some five- and six-word lines in Group I and some nine- and ten-word lines in Group II, we see that there are, in the former, more complex rhythms owing to a greater number of polysyllables. We notice also, in comparing lines with low word-count against those with high word-count, that there are generally some important syntactic differences. The lines with higher word-count have, predictably, more monosyllabic pronouns, articles, conjunctions, prepositions, and auxiliary verbs and, at the same time, seem to display a degree of syntactic independence. In other words, these lines can be read out of context without any marked loss of meaning.

Group I

Incertainties now crown themselves assured,	(107. 7)
Time's thievish progress to eternity	(77. 8)
Leaving thee living in posterity	(6.12)
Possessing or pursuing no delight	(75.11)
Creating every bad a perfect best	(114. 7)
And art made tongue-tied by authority	(66. 9)
Proving his beauty by succession thine.	(2.12)

Group II

Bare ruined choirs, where late the sweet birds sang.	(73. 4)
The hardest knife ill used doth lose his edge.	(95. 4)
When tyrants' crests and tombs of brass are spent.	(107.14)
My mistress when she walks treads on the ground.	(130.12)
That time of year thou mayst in me behold,	(73. 1)
If snow be white, why then her breasts are dun.	(130. 3)
Her love, for whose dear love I rise and fall.	(151.14)
And yet it may be said I loved her dearly,	(42. 2)
So long as youth and thou are of one date,	(22. 2)

Syntactic and metrical independence is encountered much less frequently among the lines with low word-count. Such a line is apt to be closely linked both in meaning and meter to surrounding lines, resulting more often in complex rhythms that develop beyond the confines of the ten-syllable line. A good example of this can be found in Shakespeare's sixth sonnet where we find the line "Leaving thee living in posterity"—rhythmically unusual if we compare it to the classic iambic line. In its context, however, it strikes us as being precisely what is called for:

Ten times thyself were happier than thou art
If ten of thine ten times refigured thee.
Then what would death do if thou shouldst depart
Leaving thee living in posterity.

The syntax and rhythm of the line links it inextricably to the preceding line.

The second important assumption that I make in the design of the system is that there is a definable distinction, universally applicable, between content and function words. Theoretically, function words are those that add little meaning to a sentence but are required by a language for a number of indispensable tasks. These words, or particles, will appear frequently and in almost any context, irrespective of meaning. The content word, on the other hand, is uniquely relevant to its context.

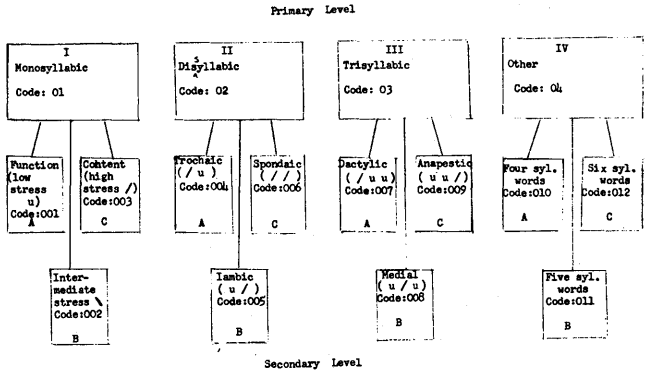
There is, alas, no clear dichotomy between the two categories. While no one can argue with the contention that the articles *a, the,* and *an* are unequivocally functional, and that their appearance has no bearing on content, a word like "some," though it might be analogous in function to an article, clearly bears an element of meaning. Furthermore, a context can radically alter our judgment about a word. To ameliorate the inevitable and inescapable problems that attend this kind of classification, I assign the monosyllables (polysyllables are treated in an entirely different manner) to three classes, thus reflecting an intermediate state between "pure" function and "pure" content word.

Monosyllables that are virtually devoid of content value are assumed in the system either never to be stressed, or receiving at most a tertiary stress, while those at the other end of the range—the content words—are assumed always to receive primary stress. I will explain more how this information is used when I get to the tagging procedure. These assumptions, plus some others that are implicit in this technique, form the basis of an automated "metrical analyzer."

The actual processing of the text begins with the development of the machine dictionaries. If computer size and speed is a seriously limiting factor, as it has been in my case with an IBM 7044, it would be impractical to use standard, universal dictionaries. Consequently, I resort to dictionaries developed specifically from the texts to be processed.

The logical organization of the dictionary is based primarily on the number of syllables in the word: Category I is for monosyllables, II for dissyllables, III for trisyllables, and IV for words having more than three syllables (See illustration). As you might expect, most of the running words in a text are monosyllabic. In Shakespeare's sonnets, for example, the 226 most frequent monosyllables account for slightly over 66% of the total occurrences of all words. Thus the lists in Category I are critical in their importance to the processor.

The organization and coding of the dictionary is as follows:



Within Categories II–IV there are secondary divisions indicating the placement of accent. Since this cannot apply to monosyllables, Category I has no such division. In Category II, subcategory A would contain the dissyllables with an accented first syllable (/u), while B would contain dissyllables with an unaccented first syllable (u/), and C would contain two stressed syllables (//). Category III has a tripartite division: words are assigned to each sub-list according to the placement of primary accent. The last category, IV, is divided only on the basis of the number of syllables. IV (A) contains the four-syllable words, IV (B) the five-syllable words, and IV (C) everything else. There is little justification because of the low frequency of Type IV words to make finer distinctions than these.

As a footnote to this, I would refer the reader to Philip Stone's discussion of the subject of dictionary development in Chapter IV of *THE GENERAL INQUIRER.*

The computer could, of course, be used for syllable counting since there are adequate programs that do this for line justification in text editing packages, but determination of stress would be, at best, difficult.

The first category of words, the monosyllables, has a secondary set of classifications based, as I explained earlier, on relative content value. Type I (A) contains the "pure" function words, I (B), the words that can be construed as having characteristics of function words with some discernible content value, and I (C), the "pure" content words. Following is a list for each of the four major types and their respective sub-types. For the sake of brevity only words having a frequency of greater than three are included. The frequency for each word on the list appears to its right. Although 2,650 of the 3,211 total words have, therefore, been omitted form the lists, the words included account for 80.2% of the total occurrences.

TYPE I (A)

1. a	169	45. of	370	91. yet	51	4. art	49
2. am	35	46. on	80	92. you	112	5. babe	4
3. an	27	47. or	81	93. your	100	6. back	9
4. and	490	48. our	19			7. bad	7
5. are	71	49. shall	59	TYPE I (B)		8. bare	4
6. as	122	50. shalt	11			9. base	5
7. at	25	51. she	33	1. ah	7	10. bath	4
8. be	140	52. should	44	2. all	118	11. bear	13
9. been	6	53. shouldst	6	3. both	17	12. best	23
10. but	17	54. since	24	4. down	5	13. birth	5
11. by	93	55. so	145	5. each	15	14. black	13
12. can	44	56. than	44	6. else	5	15. blame	4
13. canst	7	57. that	322	7. far	17	16. blind	6
14. could	11	58. the	442	8. here	5	17. blood	8
15. did	26	59. thee	161	9. how	40	18. boast	5
16. do	84	60. their	80	10. let	26	19. book	6
17. done	5	61. them	17	11. more	64	20. born	8
18. dost	30	62. then	78	12. most	27	21. brain	5
19. doth	87	63. these	21	13. much	17	22. brand	5
20. for	172	64. they	53	14. must	21	23. brass	4
21. from	82	65. thine	44	15. near	5	24. break	4
22. had	16	66. this	107	16. none	13	25. breast	7
23. hast	17	67. those	33	17. now	46	26. breath	8
24. hath	43	68. thou	235	18. 0	50	27. bright	11
25. have	77	69. though	33	19. oft	5	28. bring	8
26. he	44	70. through	62	20. once	10	29. brow	8
27. her	51	71. thus	22	21. one	42	30. buds	4
28. him	35	72. thy	272	22. out	17	31. call	10
29. his	108	73. till	14	23. own	30	32. calls	5
30. I	344	74. tis	12	24. self	85	33. care	6
31. if	68	75. to	417	25. some	31	34. catch	4
32. in	323	76. too	16	26. still	42	35. change	12
33. is	182	77. was	29	27. such	31	36. cheek	5
34. it	115	78. we	15	28. thence	8	37. cheeks	4
35. like	34	79. were	31	29. there	18	38. chide	5
36. may	29	80. when	106	30. up	16	39. child	8
37. mayst	12	81. which	106	31. well	26	40. clouds	4
38. me	168	82. while	6	32. what	75	41. cold	6
39. might	26	83. whilst	13	33. where	44	42. come	15
40. mine	63	84. who	32	34. why	25	43. count	4
41. my	392	85. whom	12			44. crime	4
42. no	79	86. whose	19	TYPE I (C)		45. crowned	4
43. nor	53	87. will	63			46. cure	5
44. not	166	88. wilt	14	1. add	4	47. dare	4
		89. with	180	2. age	15	48. date	5
		90. would	21	3. air	4	49. day	28

50. days	17	104. grow	8	158. look	22	212. sad	7
51. dead	16	105. grown	4	159. looks	12	213. said	4
52. dear	22	106. grows	4	160. lose	9	214. sake	8
53. death	16	107. hand	18	161. loss	10	215. same	6
54. deeds	10	108. hate	16	162. lost	4	216. save	8
55. deep	8	109. head	6	163. love	165	217. saw	5
56. die	13	110. hear	6	164. loved	5	218. say	28
57. doom	5	111. heart	51	165. loves	9	219. says	5
58. due	5	112. hearts	4	166. mad	6		
59. dull	6	113. heat	4	167. made	20	TYPE I (C) cont.	
60. dumb	6	114. hell	8	168. make	43		
61. dwell	6	115. help	4	169. makes	8	220. scope	5
62. earth	12	116. hence	6	170. man	5	221. scythe	4
63. end	12	117. hide	5	171. mark	4	222. sea	4
64. ere	9	118. high	4	172. men	14	223. see	35
65. eye	39	119. hold	13	173. mind	16	224. seek	4
66. eyes	51	120. holds	5	174. minds	3	225. seem	12
67. face	19	121. hope	6	175. moan	5	226. seen	11
68. faith	5	122. hour	4	176. muse	16	227. sense	4
69. false	19	123. hours	12	177 name	17	228. set	7
70. fast	5	124. hue	5	178. need	5	229. shade	4
71. faults	8	125. ill	19	179. needs	5	230. shame	10
72. fear	8	126. jewel	4	180. new	26	231. shape	5
73. fears	6	127. joy	8	181. night	22	232. shine	4
74. fell	5	128. just	4	182. oaths	4	233. short	5
75. find	16	129. keep	9	183. old	22	234. show	23
76. fire	10	130. keeps	5	184. pain	5	235. shows	6
77. first	13	131. kill	5	185. part	20	236. sick	5
78. form	11	132. kind	12	186. parts	7	237. side	5
79. forth	7	133. knife	4	187. past	10	238. sight	18
80. foul	7	134. know	17	188. pen	10	239. sin	8
81. found	10	135. knows	11	189. place	10	240. sing	7
82. frame	4	136. lack	5	190. play	5	241. skill	7
83. free	4	137. laid	4	191. please	4	242. slave	6
84. fresh	8	138. large	5	192. pluck	4	243. sleep	6
85. friend	14	139. lays	4	193. poor	15	244. slow	5
86. friends	4	140. lease	4	194. praise	28	245. smell	4
87. full	13	141. least	13	195. pride	11	246. song	4
88. gain	4	142. leave	10	196. prime	4	247. soul	10
89. gainst	6	143. leaves	8	197. proud	15	248. speak	8
90. gave	5	144. left	5	198. prove	12	249. spend	5
91. gift	5	145. lend	5	199. put	7	250. spent	6
92. give	27	146. lends	4	200. quite	5	251. spite	5
93. gives	8	147. less	7	201. rage	5	252. spring	5
94. glass	10	148. lie	13	202. rank	5	253. stand	8
95. go	6	149. lies	12	203. rare	4	254. stars	5
96. gone	9	150. life	23	204. read	5	255. state	14
97. good	13	151. light	6	205. red	4	256. stay	9
98. grace	11	152. lines	8	206. rest	6	257. steal	7
99. grant	4	153. lips	7	207. rhyme	6	258. store	9
100. great	10	154. live	29	208. rich	11	259. straight	7
101. green	5	155. lived	5	209. right	11	260. strange	6
102. grief	5	156. lives	9	210. rose	6	261. strength	5
103. groan	4	157. long	12	211. rude	4	262. strong	9

263. sum	4	315. world	28	40. image	4	91. subject	5
264. sun	11	316. worms	4	41. inward	4	92. substance	4
265. swear	7	317. worse	5	42. judgment	6	93. summer	8
266. sweet	55	318. worst	7	43. knowest	4	94. sweetest	6
267. sweets	6	319. worth	21	44. knowing	4	95. tender	7
268. swift	5	320. wound	4	45. leisure	4	96. therefore	17
269. take	13	321. writ	6	46. living	7	97. tired	4
270. taste	4	322. write	10	47. longer	6	98. tongue-tied	4
271. taught	6	323. wrong	6	48. lovely	8	99. touches	4
272. tears	5	324. year	5	49. lovest	6	100. treasure	9
273. tell	15	325. young	4	50. loving	10	101. truly	5
274. ten	7	326. youth	15	51. making	12	102. under	6
275. thief	4			52. many	13	103. very	7
276. thing	12	TYPE II (A)		53. memory	8	104. virtue	7
277. things	14			54. merit	5	105. water	5
278. think	15	1. absence	5	55. mistress	4	106. weary	4
279. thinks	6	2. absent	4	56. morrow	4	107. wherefore	4
280. thought	18	3. after	11	57. mortal	5	108. wherein	5
281. thoughts	18	4. angel	5	58. motion	5	109. whether	6
282. three	7	5. any	11	59. music	6	110. winter	6
283. time	54	6. barren	5	60. nature	10	111. worthy	5
284. times	10	7. beauties	4	61. never	16	112. wretched	4
285. time's	15	8. beauty	52	62. nothing	19	113. wrinkles	5
		9. being	32	63. numbers	4		
TYPE I (C) cont.		10. better	18	64. only	6		
		11. blessed	11	65. other	25	TYPE II (B)	
286. toil	4	12. body	4	66. others	11		
287. told	4	13. buried	5	67. outward	7	1. above	4
288. tomb	5	14. canker	5	68. painted	5	2. account	4
289. tongue	11	15. comfort	4	69. painting	6	3. again	10
290. tongues	4	16. common	4	70. pity	9	4. against	18
291. took	5	17. cruel	8	71. pleasure	11	5. alone	19
292. true	38	18. errors	4	72. power	9	6. although	9
293. truth	22	19. even	24	73. praises	4	7. antique	5
294. trust	5	20. ever	14	74. precious	6	8. appear	4
295. two	9	21. every	31	75. present	6	9. assured	4
296. use	13	22. evil	4	76. prime	4	10. away	18
297. used	4	23. fairest	5	77. public	4	11. before	15
298. verse	15	24. flower	5	78. purpose	5	12. behind	4
299. view	8	25. flowers	8	79. reason	6	13. behold	7
300. want	4	26. former	4	80. roses	7	14. believe	4
301. war	6	27. fortune	6	81. second	4	15. cannot	10
302. waste	7	28. gentle	14	82. seeing	7	16. compare	6
303. way	6	29. given	4	83. shadow	7	17. decay	9
304. wealth	6	30. glory	8	84. sinful	4	18. delight	8
305. weeds	4	31. golden	5	85. sometime	6	19. desire	11
306. white	7	32. graces	5	86. sorrow	5	20. despite	6
307. wide	7	33. gracious	5	87. sovereign	4	21. disgrace	8
308. win	4	34. happy	11	88. spirit	10	22. enough	6
309. wish	6	35. having	6	89. stolen	5	23. excuse	7
310. wit	7	36. heaven	13	90. story	4	24. forgot	4
311. woe	12	37. heavy	6				
312. words	10	38. holy	4				
314. work	4	39. honor	7				

25. forsworn 4
26. increase 4
27. removed 4
28. return 6
29. themselves 6
30. unless 6
31. upon 29
32. within 11
33. without 8

TYPE II (C)

In this illustration there are none of these words.

TYPE III (A)

1. argument 6
2. beauteous 9
3. ornament 5

TYPE III (B)

1. abundance 4
2. contented 4
3. eternal 6
4. invention 5
5. remembered 4

TYPE III (C)

In this illustration there are none of these words.

TYPE IV

In this illustration there are none of these words.

After the dictionaries are stored in memory, a record (that is, each line or sentence as the case may be) is read and each word is tagged according to its type and subtype. The tag consists of six bits of information: the first field of the tag indicates the number of words in the line or sentence, the second indicates the number of syllables in the record, the third is the position of the word in the string, the fourth is the major category or primary level code, the fifth is the subordinate category or secondary level code, and the sixth contains a number that identifies the position of the word on its respective lexicon. To illustrate, the line, "Though thou repent, yet I have still the loss" would be tagged in the following way (The blank fields are used here merely for ease of reading.):

```
09 010 01 01 001 0069  Though
09 010 02 01 001 0068  thou
09 010 03 02 005 0028  repent
09 010 04 01 001 0091  yet
09 010 05 01 001 0030  I
09 010 06 01 001 0025  have
09 010 07 01 002 0026  still
09 010 08 01 001 0058  the
09 010 09 01 003 0161  loss
```

This six field tag is sixteen character positions long. The tags are stripped from the words and concatenated with the redundant information being factored out so that the line is represented by a numerical string, as follows:

```
09 010 / 01010010069 02010010068 03020050028
         04010010091 05010010030 06010010025
         07010020026 08010010058 09010030161
```

These tags are the general pattern of the line and serve principally to provide information for statistical calculations and to provide a basis for matching with other similar strings; however, the string can be interpreted visually in such a way as to reconstruct the metrical character of the line, an entertaining pastime that is useful for testing the system and the underlying assumptions.

A more familiar notation may be substituted so that any word found in the low stress, monosyllable list (I A) will be assigned a "u", any word found in the second, intermediate list (I B) will be indicated with a "/", and a monosyllable found on the third list (I C) gets a "/". Words found in other lists would receive appropriate notation (See illustration). The line would be marked thus:

```
u     u    u /   u  u  u   \  u   /
Though thou repent yet I have still the loss
```

Because we know that words appearing on list I (A), though tending to be unstressed, receive some stress in appropriate contests, for example if two such words are juxtaposed, we can apply a simple algorithm: if two "u's" appear in sequence, the second of the two is changed to a "\". The modification would give us:

```
u     \    u /   u  \  u   \  u   /
Though thou repent yet I have still the loss.
```

The principal application of the concatenated tags, as I have noted, is to provide input for something more ambitious than this machine scansion. The basic purpose of the processor is to enable the investigator to correlate (1) the average number of syllables per word, (2) the stress pattern, (3) the positions of certain words or word types. At this writing I have not fully worked out the parameters of the statistical analysis, to wit, what kinds of tests would be most meaningful. This work, with the help of a grant from the American Philosophical Society, is presently underway. However, I think I can show in this final section of the paper some examples of how the text processing works in responding to rhythmic patterns in verse and prose, and what we can learn about style from these results.

In the accompanying illustration, only one field of the tag is used, the secondary level code, with all superfluous zeroes eliminated. The lines, from Shakespeare's sonnets, have been selected with an eye toward providing several examples of lines that are metrically similar, but at the same time with a notable point of difference. Thus we see in the two lines, "So great a

sum of sums yet canst not live," and "So long as men can breathe or eyes can see," nearly identical properties in their respective stress patterns. A likely reading would produce the following scansion:

```
u    /   u /   u /     u     \     u    /
So great a sum of sums yet canst not live
u    /   u /   u /     u     /     u    /
So long as men can breathe or eyes can see
```

The processor would have made the same distinction in its numerical notation.

```
1    3   1 3   1 3     1     1   1 3
So great a sum of sums yet canst not live
 1   3   1 3   1 3     1 3   1 3
So long as men can breathe or eyes can see
```

Six-Word Lines

```
 1  1  1    4  1   10
And by their verdict is determined          ( 46.11)
 1  1  1    4     3  10
And with his presence grace impiety         ( 67. 2)
 1     4     1  5     5     5
And therefore mayst without attaint o'erlook ( 82. 2)
 1  4  3  1  4     7
And given grace a double majesty            ( 78. 8)
1  4     4     4    4  3
If nature sovereign mistress over wrack     (126. 5)
   5  1  4     4  1  5
Against thy reasons making no defense       ( 89. 4)
   5  1  4     4  1  5
Unless this general evil they maintain      (121.13)
```

Seven-Word Lines

```
1  1  1  1  1  4     7
So dost thou too and therein dignified      (101. 4)
 1  2  1  1  8     1  5
Are both with thee wherever I abide         ( 45. 2)
 1  1 1  1     8     1  5
That I in thy abundance am sufficed         ( 37.11)
1  2 1    4     1  1  7
So all their praises are but prophecies     (106. 9)
1  1  1     4     1  5     5
So till the judgment that yourself arise    ( 55.13)
 1  4  1  1  4     1  5
The other as your bounty doth appear        ( 53.11)
 1  4  1  1  4  4     3
Thy merit hath my duty strongly knit        ( 26. 2)
 3  1 1  3  5     8     3
Kind is my love today tomorrow kind         (105. 5)
 3  4     4     1 1  4  1
Past reason hunted and no sooner had        (129. 6)
 3  4     4  1 1  4     3
Past reason hated as a swallowed bait       (129. 7)
 5     1  3  1 1  5     5
Although I swear it to myself along         (131. 8)
  5  1  3     1     4  1  5
Desire is death which physic did except     (147.12)
```

Eight-Word Lines

```
 1  1  1  3    1  1  8     3
And in this change is my invention spent    (105.11)
 1  1     1  3    1  1  8     3
Not mine own fears nor the prophetic soul   (107. 1)
1  2  1  4     1     1  4  1
For all that beauty that doth cover thee    ( 22. 5)
2  1  1  4     1  1  4  3
Ah yet doth beauty like a dial hand         (104. 9)
 2  1  1  4  1  2  4     3
All these I better in one general best      ( 91. 8)
 1     4  1  1  2  2  4     3
That music hath a far more pleasing sound   (130.10)
1     4     1  1  3  1  4  3
The wrinkles which thy glass will truly show ( 77. 5)
  3  1  1  4     1  1  4     3
Come in the rearward of a conquered woe     ( 90. 6)
  3  1  1  4     1  1  3     5
Steal from his figure and no pace perceived (104.10)
   4     1     3  1  3     4     1  3
Making their tomb the womb wherein they grew
                                            ( 86. 4)
```

Nine-Word Lines

```
 4  1  3  3  3  5  1  3
After my death dear love forget me quite    ( 72. 3)
 1  1     1  3     3  1  1  4  1
But since your worth wide as the ocean is   ( 80. 5)
 1  1  1  3  3     3     1     4  3
That in thy face sweet love should ever dwell ( 93.10)
1  2  1  3  5  1     1  1  3
And all the rest forgot for which he toiled ( 25.12)
1  1  1  4  1  1  1  1  3
As with your shadow I with these did play   ( 98.14)
 1     1  1  4     1  1  3  1  3
From whence at pleasure thou mayst come and part
                                            ( 48.12)
1  3  1  4  1  1  3  1  3
And life no longer than thy love will stay  ( 92. 3)
  2  3  1  4  1  1  3  1  3
How sweet and lovely dost thou make the shame
                                            ( 95. 1)
2  3  3  3  1  4  1 1  1
All men make faults and even I in this      ( 35. 5)
1  3  3  3  1  3  1  1  5
My love looks fresh and death to me subscribes
                                            (107.10)
```

```
  1   4     2   1   3  1  1    1  1
That beauty still may live in thine or thee     ( 10.14)
  1   4     2  1  3    1  3    1   3
That having such a scope to show her pride      (103. 2)
  3   2   1   2   1   4     1    2   3
Lose all and more by paying too much rent       (125. 6)
 3    1   1   1     5     1  1  2   3
Say that thou didst forsake me for some fault   ( 89. 1)
  5      1   3     1  1  1  1  1  2
Against my love shall be as I am now            ( 62. 1)
   5       1   3     1   3  1   3   1   3
Although she knows my days are past the best
                                                (138. 6)
```

Ten-Word Lines

```
 1   1    3  2    1   2  1  3     1   3
For no man well of such a salve can speak       ( 34. 7)
  1   1     3  1  1   3    1    2  1  3
For what care I who calls me well or ill        (112. 3)
3       1   1  1  1    2    1     1   1   3
Save what is had or must from you be took       ( 75.12)
 3      1   1    3  1  1    1    1  1   2
Save thou my rose in it thou art my all         (109.14)
 3       2    1    1   1    1     1  3  1   1
Save where thou art not though I feel thou art
                                                ( 48.10)
 3     1   1  3    1    3   1   3  1   1
Look what is best that best I wish in thee      ( 37.13)
 3   1     1   3     1  1    2    1   1  3
Kill me with spites yet we must not be foes     ( 40.14)
```

The same approach, executed on three prose passages—the first two sentences, respectively, from the "Gettysburg Address," Hemingway's *A Farewell to Arms*, and my own paper—shows results that are dramatic enough to be visible to even the most casual examination.

```
          6       1    4     3     5   1     4        3
(1) Fourscore and seven years ago our fathers brought
        2   1   1       7     1  3     4          5     1
    forth on this continent a new nation, conceived in
        7       1      10     1   1        10      1     2
    liberty, and dedicated to the proposition that all
        3   1   8        4      2   1   1      5      1  1
    men are created equal. Now we are engaged in a
          3    4   3       4          4       1    4    1
    great civil war, testing whether that nation or
        4   4      1   5          1   1     10       1
    any nation so conceived and so dedicated can
        3    5
    long endure.
    1   1    3    4      1  1     3   1   3      1  1
(2) In the late summer of that year we lived in a
          3    1  1   4     1       3      5     1    4     1
    house in a village that looked across the river and
        1    3    1  1    4          1   1    3   1   1
    the plain to the mountains. In the bed of the
        4    1       1    4      1     4         3    1
    river there were pebbles and boulders, dry and
          3   1   1   3     1   1     4     1   3     1
    white in the sun, and the water was clear and
        4      4       1   3    1  1      4
    swiftly moving and blue in the channels.
    1      4      1   1     4  1  1   5       2   1
(3) My purpose in this paper is to explain how a
           8       1  1   3   1   8          10      1
    computer can be used in prosodic analysis and
        2    1    5       1  1   3   1    7         4
    how the results may be used in stylistic studies.
        1    1    5      1   3      4     3   3    5
    When I began this work nearly two years ago,
        1    3    1   1     8     1      10            4
    my aim was to develop a computerized method
        1   10        3   1    7       1     5     1
    of classifying lines of poetry with respect to
          7          4
    rhythmical features.
```

We can see that the three passages differ significantly in several ways: in the number of low-stress monosyllables, in the number of all monosyllables, in the rapidity with which certain word types recur, etc. The sample, of course, is too small and the data too raw for any kind of meaningful observations.

Right now I am enthusiastic about this technique for text processing. Whether the approach is helpful as a tool for stylistic analysis of poetry is the first question I intend to get answered. If the results of that study augur well, refinements to the procedure and broader applications of it are in order.

An approach to the development of an advanced information management system

by J. E. MYERS and S. K. CHOOLJIAN

Butler Data Systems
Hawthorne, California

INTRODUCTION

This paper is a follow up to a presentation delivered November 19, 1969, at the Fall Joint Computer Conference, as part of the panel, "Information Management Systems for the 70's." In that panel, a prototype system, based on a unique method of logically defining data, was discussed. This paper is concerned both with what has been learned from the prototype effort, and the characteristics of a production system.

A prototype was developed using IBM/360 file management software (DL/I), demonstrated the feasibility of the logical concepts. A production system, using the same conceptual approach, has been developed by Butler Data Systems around proprietary file management software.

BACKGROUND

1970's requirements

An analysis of projected 1970's information system requirements indicated that present data management systems are incapable of providing necessary service. The following key requirements were identified:

Data definition and Data Base content must be controlled by the user.

One logical data system must handle all business data without restriction to any particular subject or data type.

Application changes must be accomplished without significant programming, resulting in reduction of incremental costs and time.

There must be a full range of information transfer (input/output) capability in both batch and interactive mode.

The information management system must be able to handle normal business data processing along with a standard interface and control for other disciplines such as numerical control or printing and publishing.

The system must be capable of relating all business data using a common data language or structure.

The system must be flexible in design such that hardware, software, and application changes cause minimum degradation of performance during implementation.

APPROACH

System concepts

The Information System is the total communication network (formal and informal) of an enterprise. It connects the various units of an organization and supports the managerial activities involved with planning, organizing, staffing, directing and controlling the firm. The Information System is composed of two major subsystems (Figure 1). The Management Information System (MIS) supplies information to users. It is composed of objectives, policies, practices and procedures dealing with its main function. The Information Management System (IMS) provides all required data processing functions in support of the MIS.

The emphasis on data processing hardware efficiency as the primary performance criterion must shift to total system effectiveness. As Figure 1 indicates, the MIS imposes performance requirements and specifications upon the IMS. The measure of response of the IMS to the MIS is the true indicator of total system effectiveness. Major system design objectives must, therefore, be predicated upon effectiveness considerations so that the IMS is responsive to user needs.

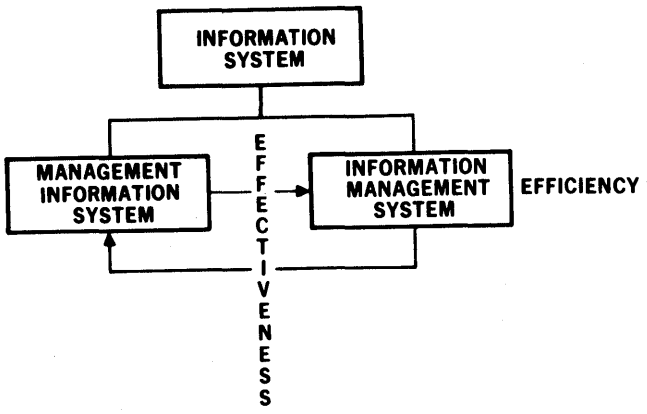


Figure 1—Information system

Management approach

Another area of emphasis must be in the management of information system development. The IMS subsystem requires a considerable amount of resources over an extended period of time. Management techniques, such as those used for development of complex military and civil systems, are directly applicable in this area, and were used in the prototype development. The system life cycle approach divides the life of a system into four phases (Figure 2).

PHASE	FUNCTION
CONCEPTUAL	Situation analysis, problem definition, conceptual approach formulated, establishment of system requirements baseline.
DEFINITION	Design trade-offs, final selection of approach, major system definition, establishment of design baseline.
DEVELOPMENT	Detailed design of all components, testing, production of operational system.
OPERATIONAL	Introduction to users and implementation, final documentation, sustaining support, final phaseout to successor system.

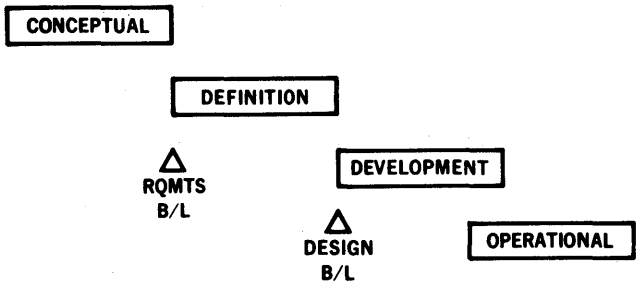


Figure 2—IMS life-cycle

The most important benefit from this approach derives from the emphasis on the first two phases. It is during the conceptual and definition phases that the users can, by actively participating in establishment of performance criteria, project management and system evaluation, assure that the resulting system will be responsive to their needs. The data system described in this paper was developed in this manner and supports an IMS that is user oriented.

Conceptual information system

The Conceptual Information System can be depicted as a series of concentric rings of activity (Figure 3). The outer ring exemplifies the MIS or total requirement for information. As shown, this requirement must be satisfied by the IMS through an information transfer function. The transfer media employed are extremely important as they provide the critical man-machine interface. Because the most important consideration in any IMS is the data, not the processing techniques or hardware employed, any approach to the development

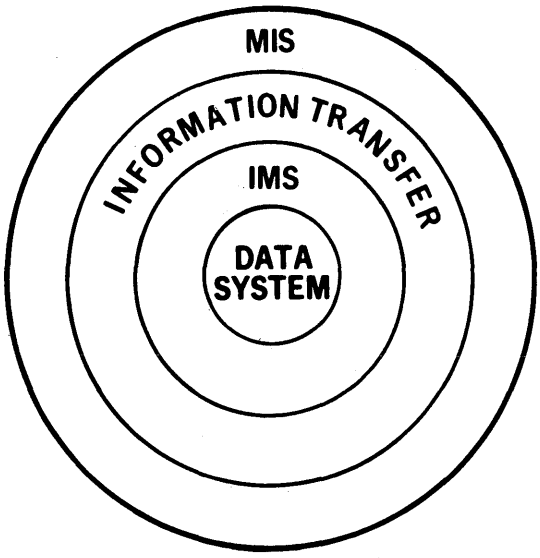


Figure 3—Conceptual information system

of a 1970's IMS must start with the data itself. Man automatically gives data meaning and order within his mind without full realization of the associative processes involved. Data processing systems by their very nature must rely on a specific method of defining and storing data to accomplish the same end. The Data System is, therefore, the most important part of any IMS as it is here that potential system capabilities or limitations are established. If data is not commonly defined to some explicit standard, and if it is not stored in a meaningful way based on this definition, its worth is greatly limited.

Traditionally the definition of data is implicit within the using computer application programs. Each program contains its own interpretation of the data it is processing, with little or no conformity between programs. The Data Division of COBOL, for example, does not provide sufficient information about the complex relationships existing among the data it identifies. This knowledge can only be obtained from the way in which the data is logically used within the procedure. Obviously this does not provide for standards of comparison even between seemingly similar application programs. Physically maintaining data with computers, even if the processing programs were programmed in a generalized way, does not attest to the data's worth or even its ability to be logically combined.

Data system

As shown (Figure 4), the Data System (Data Base) performs two distinct functions. First, it logically defines data so that it can be processed and used in a common way, and second, it stores data based on this definition. Though often so defined in the industry, a physical data storage system alone does not constitute a Data Base. A Data Base exists within limits established by the common definition of the data contained. It is imperative that Data System design begin with the data definition process.

Figure 4—Data system (data base)

Design approach

The design requirement is for a general IMS based on a common data system. In the prototype stage, a data system was developed around a method of identifying, classifying and relating data so that it can be contained in one logical Data Base, and processed with common computer programs. A prototype IMS was developed to prove the feasibility of this approach.

Prior to development of the prototype, a data survey was made by compiling a large sample of business data from a variety of sources. This data was analyzed to determine the functions performed, and the logical relationships existing among the data. From this study, a method was developed to formally classify data by function performed, and to relate functions by patterns of data interdependency. The resulting information was used to prepare a set of rules for developing the prototype.

Prototype results

The following conclusions were reached from the prototype development:

Data Base —All business data can be logically defined using a common system. One common file system can be developed around the logical data structure.

IMS Software—Common computer programs based upon the logical data structure can be developed to support all application requirements.

Systems Engineering effort after prototype development resulted in the following system architecture.

SYSTEM ARCHITECTURE

Figure 5 is a functional diagram of the total IMS. Major subsystems are:

Hardware —the computer facility.

External Service—manufacturer's hardware operating system.

Internal Service —provides system management and administrative control over all IMS activities.

Data Support —generalized processing capability to accept, maintain, retrieve, process and display data.

Application —user-defined requirements for handling data.

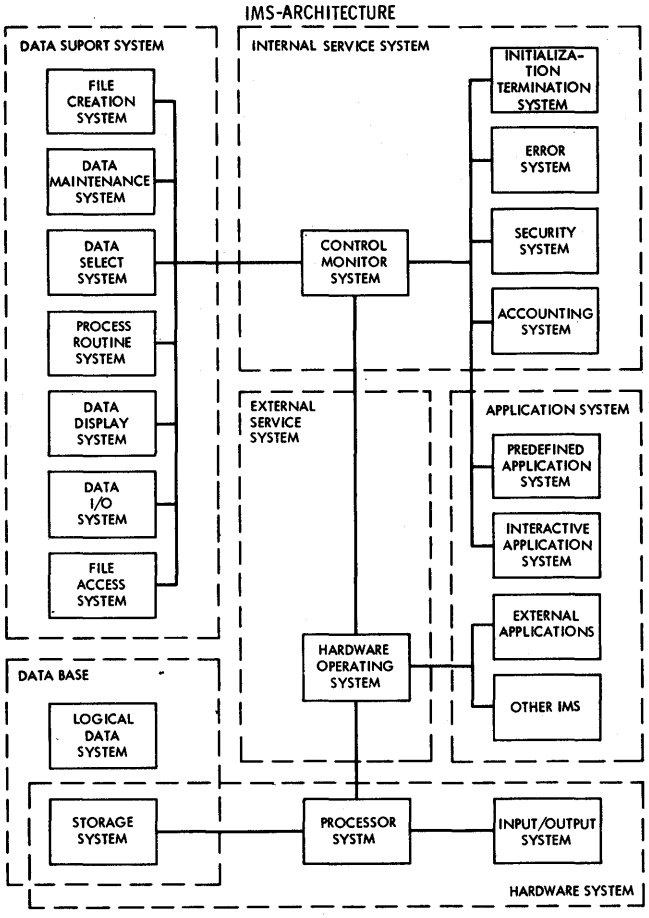


Figure 5—IMS—Architecture

Data System —logically defines and physically stores data.

The Data System is the basis around which all other subsystems are developed. Because of this, the remaining portion of this paper pertains primarily to that subject.

DATA SYSTEM DESCRIPTION

As previously stated, the Data System performs two distinct functions: the logical definition of data, and physical data storage (reference Figure 4).

Logical definition of data

The logical definition of data may be accomplished in numerous ways. However, within one IMS, only one method should be adopted. The method described herein has been generalized to accommodate any business application and, therefore, uses broad functional data classifications. This is accomplished by identifying each logical unit of data (Data Element), physically describing and functionally classifying these units, and establishing relationships between them (Figure 6).

Data Identification—The same data in various combinations are required to support numerous functions. For example, the Data Element Part Number, as associated with Labor Hours and Account Number, may be found in one application, whereas Part Number, Drawing Number, and Engineering Change Number may be associated in another. In both instances, the Part Number may be identified in common by a unique identification number (Data Element Number). The same is true of a Contract Number as it applies to an Account Number and Cost in one application, and with Engineering Change Number and Specification Number in another.

Each Data Element can be uniquely identified, independent of its many possible uses. If a data value cannot be defined in terms of an existing Data Element, a new Data Element must be identified. This is extremely important, in that data compatibility within the Data Base depends on common definitions of common data.

The exact content of a Data Element depends on the level of data control desired. For example, a date may be considered a Data Element in one instance, and a month, day, and year considered Data Elements in another. In either case, a Data Element is the lowest level at which information is defined and controlled.

The Data Element can generally be thought of as a field on a report or file record. However, these fields may be derived from combinations of Data Elements. The Data Element is to the Logical Data System what the data field is to the physical file system.

The following list of Data Elements shown in Table 1 are typical:

TABLE I—Data Element Identification

DATA NAME	ELEMENT NUMBER
Check Account Number	0001
Account Balance	0002
Overdraft limit	0003
Check Number	0004
Check Amount	0005
Trust Account Number	0006
Stock Identification	0007
Stop Loss Amount	0008
Stock Certificate Number	0009
Number of Shares	0010

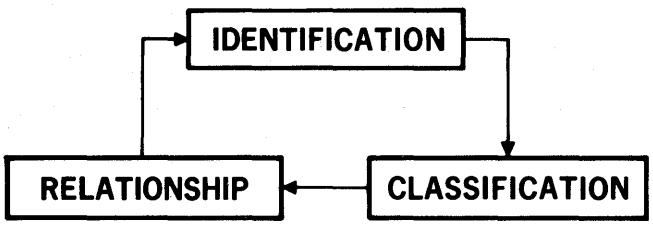


Figure 6—Data definition process

Data Classification—Data Elements can be classified by the function they perform (Figure 7). As shown, data exists either to Identify, Describe or Measure. The usual practice has been to classify business data by the organizations or applications which call for its use. These classifications are usually informal and vary with time and changes to company organization, giving rise to untold complexities of data maintenance. To avoid this pitfall, classifications must be oriented to the data functions performed.

There are two forms of identification data: The Identifier, which provides for basic forms of identification (Part Number, Man Number, Document Number, etc.) and the Modifier which modifies this basic identification (Revision Number, Serial Number, etc.). Identification data symbolically represents subjects such as the Part Number does for the hardware item or the Man Number does for an employee.

Identification data is the most important type of Data Element, because it represents company resources, products, and activities and provides the skeletal structure on which all other information rests. Few identification terms are required compared to the large amount of data usually dependent upon them.

Modifiers provide for variant forms of identification.

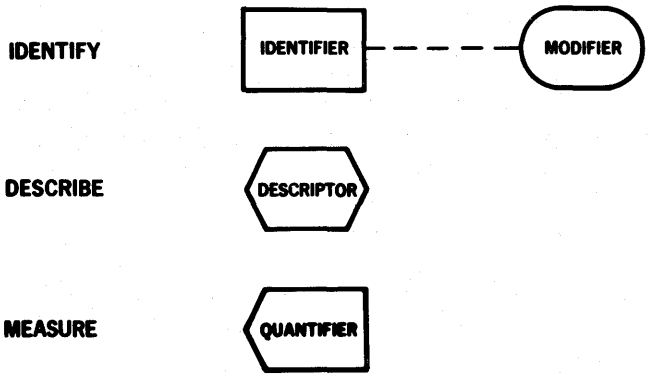


Figure 7—Data definition—Classification

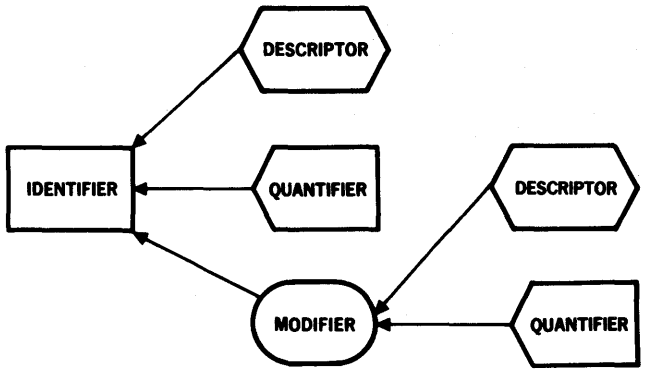


Figure 8—Data simplex

This Data Element is affixed to the Identifier and provides for unique identification of changes to, or specific occurrences of, the subject identified. The Modifier, therefore, is a constituent part of subject identification.

Descriptors are those Data Elements which are used to describe subjects. This is usually accomplished by use of titles, synonyms, abstracts, descriptions, etc. Descriptors are usually unformatted and may be one character, a complete document text or a digitized illustration. This data element type is of prime importance to the merging of data management and printing/publishing technologies.

Quantifiers provide measurement or condition (time, cost, technical) of identified subjects and are computational in format.

This functional classification technique was validated by classifying thousands of Data Elements in a broad variety of application uses.

Data Relationship—The described data classifications relate in a very definite way. The Descriptor and Quantifier depend directly on the Identifier or upon some Identifier/Modifier combination (Figure 8). The combination of all Data Elements as they relate to one subject identification can be considered a Data Simplex.

In the illustrated Data Simplex (Figure 9), the

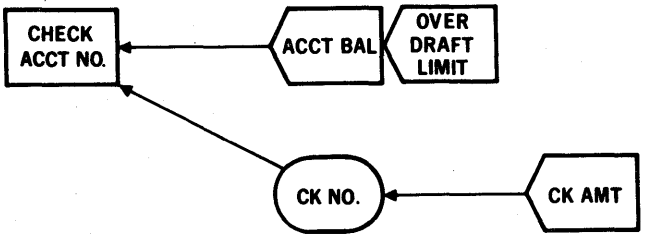


Figure 9—Data simplex—Check account

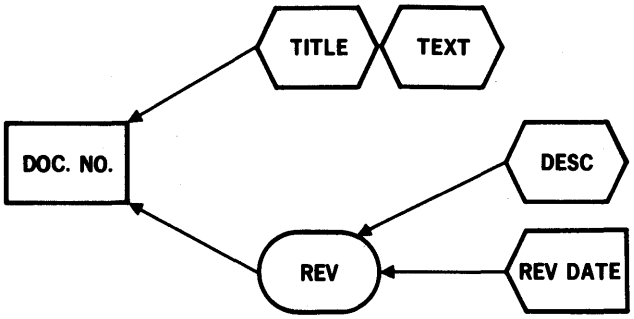


Figure 10—Data simplex—Document Number

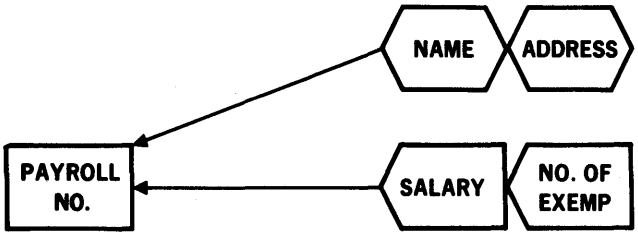


Figure 11—Data simplex—Payroll number

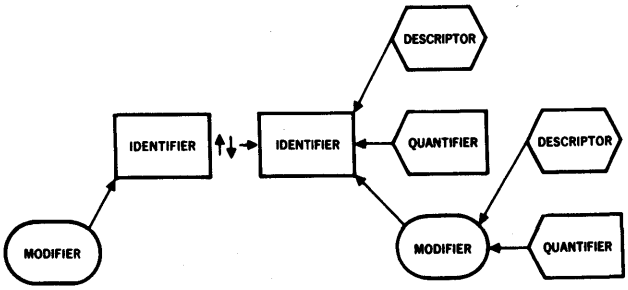


Figure 12—Data complex

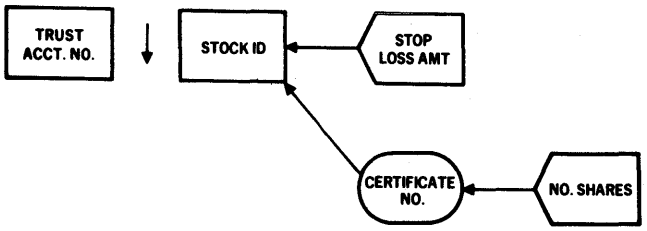


Figure 13—Data complex—Trust account number

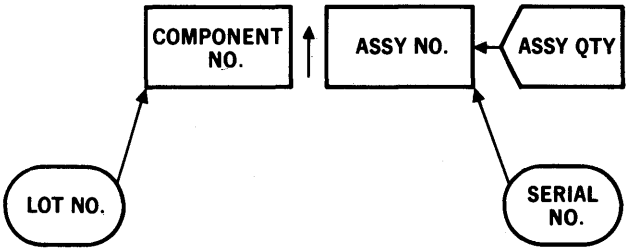


Figure 14—Data complex—Component number

Identifier is the Checking Account Number with two Quantifiers, an Account Balance and an Overdraft Limit. The Check Number modifies the Checking Account Number and the Check Amount ties to the Checking Account Number through the Check Number.

In Figure 10, the Identifier is a Document Number; tied directly to it are two descriptors, Title and Text. The Document is modified by a given revision, and attached logically to that identification are a revision description and a quantifier, Revision Date.

In Figure 11, the Data Simplex consists of a Payroll Number and two Descriptors, Employee Name and Address. In addition, there are two Quantifiers, Employee Salary and Number of Tax Exemptions.

Many Data Elements only have meaning as they apply to multiple subjects. As shown (Figure 12), two identification terms may be related in either a hierarchical or equal fashion; these dependencies present complex situations of information relationship. Descriptors, Quantifiers, and Modifiers may only have meaning as they relate to two Identifiers or Identifier/Modifier combinations.

This relationship of data, as it applies to multiple identification terms, is considered a Data Complex; most interdisciplinary data falls within this category. An example of the Data Complex (Figure 13) is a trust Account Number, and a related stock portfolio, represented by a Stock Ticker Symbol. Tied to this Identifier combination is a Stop Loss Amount. Further, the Stock Ticker Symbol can be modified by an individual Stock Certificate Number, and associated to the Number of Shares Owned.

In Figure 14, a component is shown as dependent to an assembly. This Identifier relationship is modified by lot and serial numbers. Further, this identification combination has an associated Quantifier, Assembly Quantity. The Assembly Quantity has meaning only as it applies to both Identifiers.

Two Identifiers may be considered equal (Figure 15). In this example, FICA is equal to Payroll Number, both

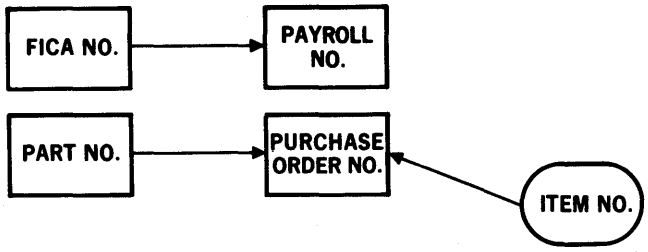


Figure 15—Data complex—FICA number, part number

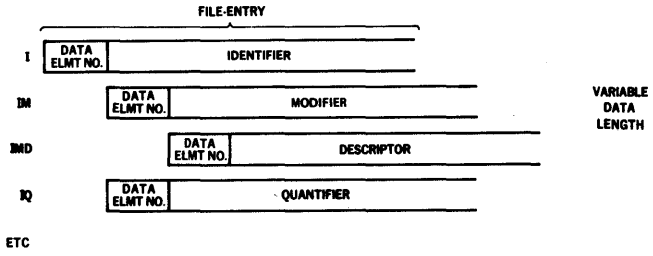


Figure 16—Physcial file organization

representing a person. Also, a Part Number may be equated to a given Purchase Order Item.

Logical Data Structure—There is a finite number of data relationships derived from the various combinations of identification description and measurement classifications. These can be expressed in symbolic form.

SYMBOL	DATA CLASSIFICATION
I	Identifier
M	Modifier
D	Descriptor
Q	Quantifier

Each relationship can be represented by combinations of these symbols; i.e., "IMQ" describes an Identifier/Modifier/Quantifier relationship (shown in Figure 9) using Check Account Number/Check Number/Check Amount.

The complex relationship "I $\downarrow$ IQ" is shown in Figure 13 using Trust Account Number, Stock Identification, and Stop Loss amount.

Data Element numbers, as previously identified in Table I, can be added to these symbolic relationships to identify a unique logical data element structure. For example:

$$I(0001) \cdot M(0004) \cdot Q(0005)$$

or $$I_1(0006) \downarrow I_2(0007) \cdot Q(0008)$$

Any logical relationship of data can be expressed by using the above techniques. This means that a physical file system based on these symbolic relationships can support any data.

Physical data storage

The physical file is a translation of the logical data structure on to a data storage device.

File Organization—There are many file organization schemes which could satisfy the foregoing requirements (i.e., Ring Structure, Inverted List, Multi-list or Indexed Sequential). The physical file system described uses two data organization methods. The first two is a hierarchical structure of indexed sequential organization used for primary data storage; the second is random. The direct address from the random file is stored as data in the primary file. This provides for multiple levels of data storage. This scheme was chosen because of its ability to support both sequential accessing for batch processing and direct accessing for interactive processing.

Depending on storage requirements, data may be stored by either method. For example, a Document Accession Number (Identifier) would be stored in the primary file. The full text (Descriptor) may be in secondary storage. The address of the text would be stored with the Accession Number.

Record Definition—Each record is composed of data relating around one subject Identifier. Each Identifier occurrence is an indexed record key or file entry. Dependent Data Elements are placed at subordinate hierarchical positions (Segments) depending on their relationships (see Figure 16).

Multiple segment types are allowed at each of the dependent levels and there can be multiple occurrences of each. Superior segment values are implied in the meaning of subordinate segment values.

Each Segment contains only one Data Element type; i.e., Identifier, Modifier, Descriptor or Quantifier.

Data Elements may be variable in length. Storage is not provided unless data is present, thus resulting in efficient storage utilization.

Data Elements stored in the same relationship are differentiated by storing a binary coded Data Element Number within the value.

The data file is composed of a fixed number of logical data combinations. As a result, processing routines can be programmed around these relationships in a highly specialized manner to optimize data processing efficiency.

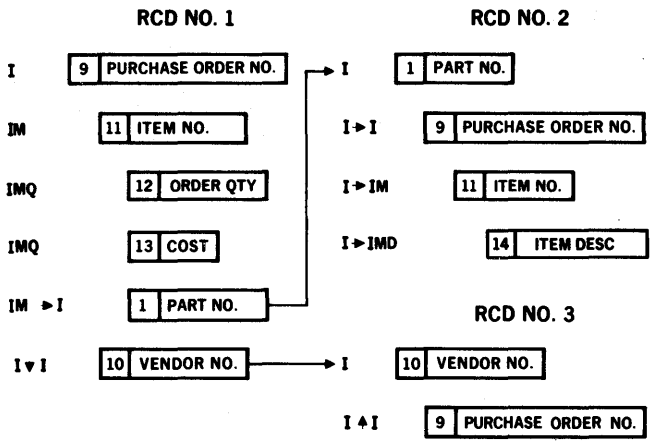


Figure 17—Physical file—Example

In the following example (Figure 17), three subject records are represented. Record 1 is a Purchase Order Number with dependent Data Elements: Item Number, Order Quantity, Cost and Part Number. The dependent Identifier, Part Number is indexed to record number 2 where it itself is an indexed key. Likewise, the Vendor Number indexes to record number 3. In record 2, the dependent Data Element, Purchase Order was automatically created (logical opposite) and points back at record 1.

As shown, the logical opposite of the Purchase Order, Purchase Item Number and Part Number (which is an "IM → I" relationship) is the opposite of the "I → IM" relationship shown in record 2. The same is true of record 3. The result of this approach to file organization, is data-oriented files in which each subject and related Data Elements are identified, with subject linkages automatically created and maintained. The file is, therefore, organized around self-indexing subjects and not around unrelated application programs. This permits the sharing of common data and reduces duplicate information. Most of all, data compatibility and relatability are insured for interdisciplinary activities.

APPLICATION CONCEPTS

Several key application concepts result from this approach to IMS development.

User interface

The worth of this approach can be demonstrated by describing the way a user interacts with the system.

DICTIONARY

DRAWING SIZE CODE [0212] An alphabetic character A thru K designating the finished size of a drawing as defined by Table 1 of MIL-STD-100.

EMPLOYEE NUMBER [0110] The number assigned to a person upon hire. Used as employee identification for personal accountability purposes. *SYN.* **MAN NUMBER, PAYROLL NUMBER.**

PART NUMBER [0102] *ABBR.* **PN,** The number used to identify, in common, all Parts, Components or Assemblies that are interchangeable in all applications where used. Its prime function is to control hardware assembly and replacement on the basis of interchangeability. *SYN.* **COMPONENT, ASSEMBLY.**

SOCIAL SECURITY NUMBER [0152] *ABBR.* **SSN.** The government assigned number for an individual registered by the Federal Insurance Contribution Act. Often used for employee identification.

Figure 18—Data element dictionary

The user defines his data (irrespective of application) to the IMS by using a single data definition system. This results in a data element dictionary (Figure 18) automatically maintained and produced by the system.

The user logically relates data elements as part of the data definition process. This places control of file content in the hands of the user and results in a completely flexible file of user design. A file definition report is produced by the IMS. The combination of the two reports gives complete data visibility within the system.

Data can be redefined with the same procedures used in original definition; this is accomplished by direct entry to the system. The elimination of programming effort is achieved because data definition is not part of the procedural portion of data processing modules, as in orthodox applications oriented systems.

The source and format of the data are defined to the system through a series of declarative statements. This allows automatic reformatting of input to the predefined IMS logical structure via a standard transaction format. A common set of file management programs automatically perform storage and update functions.

Data retrieval criteria and display characteristics (device and format) are defined to produce data output.

Special processing programs can be specified during any phase of the processing stream.

Data oriented files

Because the physical file is data-oriented, or physically structured around the dependencies of the data contained, it can support any application. Data is physically stored the way in which it has meaning to the user.

An example of possible benefits is in the mechanization of product definition (parts) data, which must be maintained for many purposes. The same data elements are often required for manufacturing, planning, tooling, material control, purchasing, provisioning, product support and publication purposes, as well as Project Management requirements for configuration management and contract estimating. Typically, parts data is independently mechanized by the organizations supporting these functions. The following are the usual results of this approach:

—Data does not agree with one another
—Data does not interrelate with one another
—Duplicate data often exists to satisfy similar objectives
—Data does not collectively satisfy product definition requirements.

The scope of the problem is immense, considering that seventy to eighty percent of the information stored is identical. By establishing data-oriented files, the above problems can be eliminated.

The application-oriented data files shown in Figure 19 contain redundant parts information.

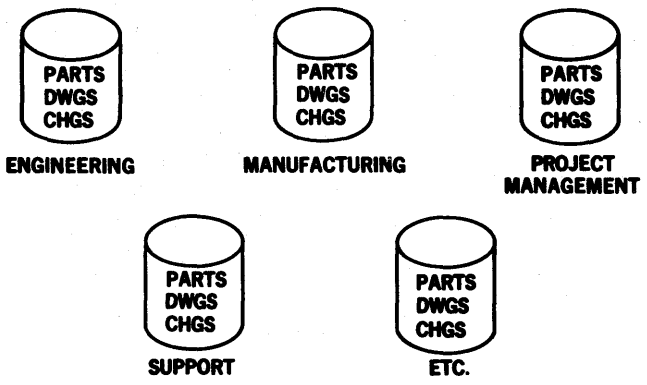


Figure 19—Application oriented files—Example

Figure 20—Data oriented files—Example

Figure 20 shows the same data elements as maintained using a data-oriented approach.

Interdisciplinary considerations

In addition to benefits gained by sharing physical files, the system provides a common data communication language. This means that transfer of data among different users is carried on automatically within the system using the same physical data and common definitions. This eliminates the usual problem, such as disagreements on content of data, timeliness of multiple data input reports, and sequence or phasing of information.

Even though multiple users may share the same data, IMS applications can be independently implemented, since the Data Base is application independent.

Because all data relationships were established in the initial file design and the IMS does not assume any predefined data content, applications can be incrementally implemented without reprogramming.

Transfer media

IMS applications are not limited to any single type of transfer media.

The IMS can handle full text and illustrations. This allows printing/publishing directly from the Data Base, using photo composition equipment. For example, Figure 18 was produced in this manner, using RCA 830 VIDEOCOMP equipment. This provides a practical integration of data management and printing technology.

SUMMARY

There are a large number of deficiencies in present Information Management Systems, primarily in their effectiveness to the user. These problems basically stem

from poor data definition and the inability to respond to user requirements. The conceptual approach to the solution of these problems was to organize the IMS around the logical definition of data. In order to prove this concept, a prototype system was designed and tested. The concept that all data can be logically defined in common has proved feasible and is the basis for the described production system.

GENERAL REFERENCES

G DODD
Elements of data management systems
Computing Surveys ACM Vol 1 No 2 1969

J D ARON
Information Systems in perspective
Computing Surveys ACM Vol 1 No 4 1969
Information management system/360—program description manual
International Business Machines Corp.
Code 6185 Program Code 360A-CS-31X
IBM/360 direct access storage devices and organization methods
International Business Machines Corp C20-1649
GE-625/635 integrated data store
General Electric Co CPB-1093A

J GOSDEN E RAICHELSON
The new role of management information systems
Mitre Corp Data Management Series No 3 MTP-332
Systems engineering management
Air Force Manual AFSCM-375-5
Product definition in the aerospace industry
International Business Machines Corp E20-0235

The dataBASIC language—A data processing language for non-professional programmers

by PETE C. DRESSEN

General Electric Company
Phoenix, Arizona

INTRODUCTION

The dataBASIC*Language is a terminal oriented data processing language which combines data base manipulation capabilities with a BASIC[1] like procedural statements. The language is designed for use by a non-professional programmer. Emphasis is placed upon simplicity and ease of learning and remembering its syntax and rules. The language permits the storage, maintenance, retrieval, and output of data on a content addressable basis. Records of any size, containing from one field to hundreds of fields may be created without record descriptions. Records are self-defining and are processed on the basis of the field names and associated field values supplied by the user at the time of record storage. Records are retrieved for maintenance or output based on the selection logic specified by the user. Records selected are processed one at a time in accordance with processing statements specified.

The dataBASIC Language is designed for operation in a general purpose time sharing system environment. As such, it is assumed that the command language of time sharing system is used to perform certain functions. These include sign-on sequence, constructing and saving programs written in the dataBASIC Language, allocating and deallocating named-file space for dataBASIC files and invoking the dataBASIC Language Processor.

DATA DESCRIPTIONS

For simplicity, record, field, and set declarations have been removed from the language. Records, fields and sets are, however, very much part of the fundamental concepts taught to and used by the dataBASIC programmers, all handled procedurally.

Records in the dataBASIC Language have no predefined format or content. They consist merely of the collection of fields which they currently hold.

The record itself has no specific name and is identified only on the basis of content. Records stored within a specific file form a structure size, which in terms of total records and record fields, is restricted only by the amount of physical space allotted for the file by the user. An important concept in the dataBASIC Language is that of a "current record," since many statements in the language are dependent upon the presence of a "current record" for execution. Only two statements in the language make a record the "current record"; the STØRE RECØRD statement which creates a new record and makes it the current record, and a FØR *relational:expression* statement which selects and makes current, one at a time, all records satisfying the specific selection criteria.

A field is a combination of a field name and its associated field value (a field name/field value pair). A field name is the name of some "thing" or attribute that can be measured or given value and a field value is a representation of that measure. The user of the dataBASIC Language is free to select and establish any number of different field names and field values. Furthermore, any number of field name/field value pairs may be associated with a given record. Fields associated with a record need not be unique, that is, there may exist occurrences of duplicates. Likewise, a file may contain any number of duplicate records. The existence or nonexistence of duplicate fields, as well as records, is controlled procedurally by the user, as is the presence or absence of records in a file and fields in a record.

Sets and set classes[2] are fundamental to many aspects of data base management, however, because of many alternate implementations and diverse uses, they have not always been clearly recognized. In the case of the dataBASIC Language, the rules for set membership are defined procedurally[3] rather than statically under

* Trademark of the General Electric Company

data declaration control.[4,5,6] The dataBASIC Language's FØR *relational: expression* statement provides this set creation and manipulation mechanism. Records are selected and presented, one at a time, as the "current record" to be acted upon. Such set generation capabilities give the advanced dataBASIC programmer data structuring tools of considerable merit.

DATA MANIPULATION CAPABILITIES

A dataBASIC record is created when a STØRE RECØRD statement followed by one or more STØRE *field* statements are executed. The STØRE RECØRD statement establishes a new record and makes it "current." Once a current record is established, the STØRE *field* statement may then be used to affix fields to the current record.

Figure 1 illustrates the language statements used in the creation of a record. The statements, when executed, will create a record. The record, for the moment, has six fields with three of the fields having the same field name, but different field values.

```
100 STØRE RECØRD
110 STØRE PRØJECT "DATABASIC"
120 STØRE FUNDS "ASD"
130 STØRE BUDGET 10000
140 STØRE ASSIGNED "JØNES" "ALLEN" "YOUNG"
```

Figure 1

Selection of records from a dataBASIC file is accomplished by using a FØR *relational: expression* statement. The *relational: expression* is used to specify the selection criteria and records selected for processing are those records in the file whose data field content are consistent with the relational:expression specified. The FØR *relational:expression* statement initiates the accessing process and, with the corresponding NEXT statement,* de-limits the processing of the current record. When the corresponding NEXT statement is encountered, the record is no longer current and the next record is selected. After all records meeting the conditions specified have been selected, control passes to the statement following the NEXT statement. Upon execution of the statements in Figure 2, all records containing the field PROJECT "DATABASIC" and the field FUNDS "ASD" will be selected, one at a time, and made available for processing.

* The dataBASIC system assigns the "corresponding NEXT statement" by pairing each NEXT with the immediately proceeding FØR statement for which there is no associated intervening NEXT statement.

```
100 FØR PRØJECT = "DATABASIC"
110 AND FUNDS = "ASD"
      .
      .
      .
200 NEXT
```

Figure 2

The dataBASIC Language also allows the user to select field names and field values *independent* of record selection. To accomplish this, two additional forms of the FØR statement are introduced: a FØR FNAME statement is used for accessing field names and a FØR FVALUE statement is used for accessing field values. A field value may be accessed and made current only after a field name has been accessed and made current. Once accessed, the field names and field values are available for processing or display. In Figure 3, each unique field name in the file will be made current for processing and for each field name made current, each associated unique field value would be made current.

```
100 FØR FNAME ALL
110 FØR FVALUE ALL
      .
      .
      .
200 NEXT
210 NEXT
```

Figure 3

A user desiring to modify the content of a current record may do so by using a STØRE *field* statement, FIX *field* statement or DELETE *field* statement. These statements allow the addition of new fields, the modification of existing fields and the deletion of existing fields. In addition, a DELETE RECØRD statement is provided to remove a current record from the file. After execution of the DELETE RECØRD statement, there is no longer a current record available for processing. Figure 4 shows a set of statements, the execution of which will cause the following: for each record in the file with a field PRØJECT "SPECIAL," the field FUNDS "ASD" will be deleted, the field BUDGET 1000 will be modified to read BUDGET 15000 and the field FUNDING "DPØ" will be added.

```
100 FØR PRØJECT = "SPECIAL"
110 DELETE FUNDS
120 FIX BUDGET = 15000
130 STØRE FUNDING "DPØ"
140 NEXT
```

Figure 4

A LET statement together with working storage fields allows a user to temporarily hold and manipulate values of fields during program execution. Values may be assigned as a result of data movement, expression evaluation or function evaluation. The content of working storage is completely controlled by the user except for the SUM, MIN, and MAX functions. For these functions, both initialization and assignment are accomplished by underlying procedures. Execution of the statements in figure 5 would result in the sum of all budget amounts for DPØ to be placed into working storage location A.

```
100 FØR FUNDS = "DPØ"
120 LET A = SUM BUDGET
130 NEXT
```

Figure 5

Users may display the entire content of the current record, selected fields of the current record, the content of working storage locations, literals, or any combination thereof. Format control may be completely controlled by underlying procedure, partially controlled by underlying procedure and partially by the user, or completely by the user depending upon the options specified. If a user wished merely to print the entire record and does not desire to control the output format, he uses a PRINT RECØRD statement as illustrated in Figure 6.

```
100 FØR PRØJECT = "DATABASIC"
110 PRINT RECØRD
120 NEXT
Output upon execution of statements might be
    PRØJECT "DATABASIC"  FUNDS "ASD"
BUDGET 10000 ASSIGNED "JØNES", "ALLEN", "YØUNG"
```

Figure 6

However, if a user desires a specific format and only selected fields to be displayed he may accomplish it with statements similar to those used in Figure 7.

```
100 PRINT "PRØJECT        BUDGET"
110 PRINT
120 FØR FUNDS = "ASD"
130 PRINT PRØJECT EDIT "BBXXXXXXXXXXBB";
140 PRINT BUDGET EDIT "99,999"
150 NEXT
Output upon execution of statements might be
                PRØJECT    BUDGET
              DATABASIC    10,000
```

Figure 7

SUPPORT FUNCTIONS

In addition to the dataBASIC statements discussed thus far, there are certain statements in the dataBASIC Language which play an important support role. These statements are especially useful when attempting to solve more complex problems. Capabilities provided by the introduction of the statements into the language include provisions for writing recursive subroutines and for the conditional restoring of a file.

Recursive subroutines are quite valuable when dealing with complex data structures like tree or network structures. A dataBASIC procedure becomes a subroutine only when accessed through the execution of a GØSUB statement. A subroutine is terminated when a RETURN statement is encountered in the sequence statements being executed as a subroutine. Control then returns to the statement following the GØSUB statement which invoked the subroutine. Unlimited subroutine recursion in the language is provided by allowing the user to specify as part of the GØSUB statement working storage fields to be treated as variables local to the subroutine. When the GØSUB statement is executed, the current value of each working storage field specified is saved and a null value assigned. When returning from the called subroutine each saved value is restored and intermediate values are lost. The use of recursive subroutines is further explored in the discussion of processing techniques.

The dataBASIC Language gives the user the ability to restore a file to the status prior to the beginning of the current run. This allows the user to debug new programs against an established data base as well as check file modifications for accuracy while having the freedom to restore the file, should an error in data or logic be detected during the run. File restoring is specified by inserting a RETREAT statement at desired locations in the program. When the RETREAT statement is executed, the program is terminated and the file restored. In the case where an unrecoverable system error occurs, file restoring is automatic.

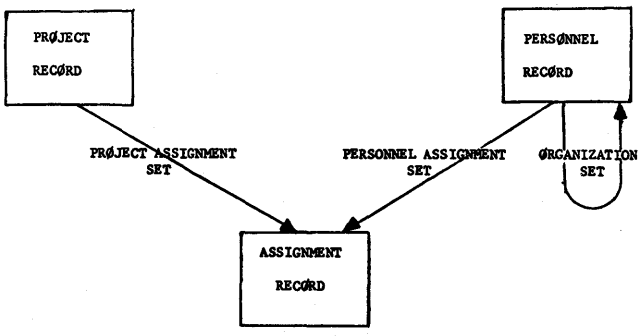


Figure 9

```
100  STORE  RECORD
110  STORE  PROJECT "DATABASIC"
120  STORE  FUNDS "ASD"
130  STORE  BUDGET 10000                    } PROJECT RECORDS
140  STORE  RECORD
150  STORE  PROJECT "SPECIAL"
160  STORE  FUNDS "ASD"
170  STORE  BUDGET 15000

180  STORE  RECORD
190  STORE  ASSIGNMENT "DATABASIC"
200  STORE  ASSIGNED "JONES"
210  STORE  RECORD
220  STORE  ASSIGNMENT "DATABASIC"
230  STORE  ASSIGNED "ALLEN"                } ASSIGNMENT RECORDS
240  STORE  RECORD
250  STORE  ASSIGNMENT "SPECIAL"
260  STORE  ASSIGNED "MASON"
270  STORE  RECORD
280  STORE  ASSIGNMENT "SPECIAL"
290  STORE  ASSIGNED "SMITH"

300  STORE  RECORD
310  STORE  NAME "SMITH"
320  STORE  SALARY 1500
330  STORE  MANAGER "YOUNG"
340  STORE  RECORD
350  STORE  NAME "JONES"
360  STORE  SALARY 1350
370  STORE  MANAGER "YOUNG"
380  STORE  RECORD
390  STORE  NAME "ALLEN"                    } PERSONNEL RECORDS
400  STORE  SALARY 950
410  STORE  MANAGER "JONES"
420  STORE  RECORD
430  STORE  NAME "MASON"
440  STORE  SALARY 1000
450  STORE  MANAGER "JONES"
460  STORE  RECORD
470  STORE  NAME "YOUNG"
480  STORE  SALARY 2000
490  STORE  MANAGER "WILLIAMS"
```

Figure 10

PROCESSING TECHNIQUES

To a casual user of the dataBASIC Language, the preceding discussion of data manipulation capabilities should suffice. However, a more serious programmer should find an examination of some of the powers of the dataBASIC Language to treat with complex data structure interesting.

Consider the Data Structure Diagram[2] in Figure 9 which represents the data structure as seen by the file user for a project control application.

Depicted is a compound network structure and a tree structure. In the compound network structure there is a class of project records (class meaning potential to exist), a class of personnel records and a class of assignment records. For each actual project record, there is a set of assignment records, each assignment record associating the project with one personnel record representing the person assigned. In the tree structure there is one class of records, personnel records, and one set class which represents an organization relationship.

Assume for the moment that the user wishes to deal only with the data items project name, funds, budget, employee name, salary, and manager. By placing the project name, funds, and budget in the project record; the employee name, salary and manager in the personnel record; and by adding assignment and assigned, and placing them in the assignment record, the user will be able to create, through procedures, the data structures desired. Let's now examine a specific case of such a file and see how it might be used.

By executing the statements in Figure 10, a file is created with records belonging to the classes discussed.

```
100 FØR FUNDS = "ASD"
110 LET A = PRØJECT
120 PRINT PRØJECT
130 FØR ASSIGNMENT = A
140 LET B = ASSIGNED
150 FØR NAME = B
160 PRINT "    " NAME SALARY
170 NEXT
180 NEXT
190 NEXT
```

Output resulting from execution of the above statements would be:

```
"DATABASIC"
    "ALLEN" 950
    "MASON" 1000
"SPECIAL"
    "JØNES" 1350
    "SMITH" 1500
```

Figure 11

Let's now assume that a user wishes to print a list by project of persons assigned to the projects funded by ASD together with their salaries. The fulfillment of this request requires a network structure with relationships like those shown in Figure 9. The dataBASIC Language statements in Figure 11 will provide this, therefore, they represent the procedure required to create such a structure dynamically. On close examination of the statements it can be seen that data resulting from selection specified by statement 130 is used as the selection criterion in statement 150. This action, termed "pivoting," allows for a closed feedback loop within a program and is an often necessary part of network processing.

```
100 LET C = 0
110 LET A = "WILLIAMS"
120 LET B = A
130 FOR MANAGER = B
140 LET C = C + 1
150 LET D = C
160 PRINT "      ";
170 LET D = D - 1
180 IF D > 0 THEN 160
190 PRINT NAME
200 PRINT
210 LET A = NAME
220 GOSUB 120 B
230 NEXT
240 LET C = C - 1
250 RETURN
```

Output when the procedure is executed would be

```
"YØUNG"

    "JØNES"

        "MASØN"

            "ALLEN"

    "SMITH"
```

Figure 12

As a second problem, consider the printing of the organizational structure of all persons under Mr. Williams' control. This will require the creation of the tree structure which defines organizational relationships. The statements in Figure 12, upon execution, represent a solution to the problem. Introduced is the use of a recursive subroutine for dealing with the tree structure.

SUMMARY

A terminal oriented data processing language has been defined and some of the more interesting capabilities discussed. The language is designed to minimize learning time. In supporting this concept, all data descriptions have been removed and are handled procedurally. In its basic form, the dataBASIC language is a procedural language for on-line record storage, retrieval, and display. Additional capabilities such as field name and field value manipulation, conditional file restoring, and recursive subroutines are provided for the more proficient user. This paper has discussed only those aspects of the dataBASIC language which are visiable to the user. Underlying data structure and procedure[7] required to implement the language capabilities, including on-line update and retrieval and data descriptors defined during execution of the program are subjects we leave for discussion at a later date.

REFERENCES

1 *Basic language reference manual*
IPS-202026A Rev Information Services Department
General Electric Company January 1967

2 C W BACHMAN
Data structure diagrams
Data Base Quarterly of SIGBDP Summer 1969

3 C W BACHMAN
The DataBASIC System—A direct access system with dynamic data structuring capabilities
Internal Paper 1970

4 C W BACHMAN S B WILLIAMS
A general purpose programming system for random access memories
Proceedings of FJCC San Francisco California October 1964

5 G G DODD
APL—A language for associative data handling in PL/I
FJCC 1966

6 *Data base task group report*
Made to the CODASYL Programming Language Committee October 1969

7 R S JONES
Search path selection for inverted files
Internal Paper 1970

LISTAR—Lincoln Information Storage and Associative Retrieval System*

by A. ARMENTI, S. GALLEY, R. GOLDBERG, J. NOLAN and A. SHOLL

Massachusetts Institute of Technology
Lexington, Massachusetts

INTRODUCTION

This paper describes an information storage and retrieval system called LISTAR (**L**incoln **I**nformation **S**torage and **A**ssociative **R**etrieval) being implemented on Lincoln Laboratory's IBM 360/67 computer to run under the IBM CP/CMS time-sharing system. An experimental version of LISTAR designed to test its main features was implemented on the IBM 7094 under the MIT Compatible Time Sharing System (CTSS). This version was described in some detail in Lincoln Laboratory Technical Report 377.[1] Because of its experimental nature, the CTSS version of LISTAR limited the total space for data files to the non-program space available in core memory. The current version allows the file space to extend beyond core memory to auxiliary storage. The logical limit of this space is determined by the addressing capacity of the system. File space size is currently fixed at 2^{30} (1000 million) bytes. This could be increased by relatively minor program changes. The current version also introduces new techniques in space management to deal with this extended file space.

LISTAR is written almost entirely in FORTRAN in order to render the system somewhat machine independent. The basic input-output routines and a small number of other basic programs were written in assembly language.

GENERAL FEATURES

LISTAR is primarily an on-line interactive system which permits a user to define, search, modify, and cross associate data files to suit his own special interests and needs. The system assumes an open-ended library of users' files which can be called from auxiliary storage for processing by a name designation. A collection of files which have a common directory (Master File) is called a "file set." Each file set contains, in addition to its data files, the directory information required for their interpretation and processing. The user is free to define, modify, augment, delete, and cross associate data in files as his interests dictate. In addition to direct support of the terminal user, the system allows information storage and retrieval functions to be performed at the request of independent task programs.

A file in LISTAR is a set of entries. Each entry consists of a set of data fields which describe the objects covered by the file name. For example, if the subject matter of the file is books, then the data fields might be title, author, date of publication, publisher, and so forth. The data fields ascribed to a file apply to every entry in the file. The number of data fields per entry, the number of entries per file, and the number of files per file set are all variable and the user is free to define as many as he wishes. Data field values may be any character string, an integer, a floating point number, etc., or a list of any one of these. Entries are ordered on the data value in a user specified field called the chief field. Information which describes the structure of files is maintained in a directory called the Master File. The Master File is structured like any other file in the system and contains information describing all files including itself.

The system has been designed to permit the user to create an association between any two files or parts of files in a file set by defining a relation between them and giving it a name. This is a system feature which was implemented and tested in the earlier version of LISTAR and is being implemented in modified form for the current version. A relation associates each entry of the first file, called the "parent" file, and a set of

*This work was sponsored in part by the Department of the Air Force and by the Public Health Service, Department of Health, Education and Welfare.

entries from the second file, called the "linkee" file. An entry from the parent file is called the "parent" entry, and the subset of entries from the linkee file with which it is associated is called its "subfile". The user must also explicitly identify the entries in the linkee file which are to be associated with each entry of the parent file.

When a relation is defined between two files, the system associates the subfile entries according to some ordering rule on one of the data fields of the subfile. The user is asked by the system to give the ordering rule and the field on which the ordering is to be done.

The ability of a user to relate files in the system and to search on these relations is one of the powerful features of the system. It gives the user great latitude in establishing and using cross references between files. The user is free to create as many files and as many relations as he chooses. He can therefore cross reference the same files in many different ways if this serves his purpose. Each relation is independent of the others. A file may take part in any number of different relations either as parent or linkee. The same file can be both parent and linkee in a relation. Multiple users can define relations on the same file set without disturbing the data base.

LISTAR SUPERVISOR AND COMMAND LANGUAGE

The supervisor program for LISTAR, called the "Resident Interpreter," is a Fortran main program. All input to the LISTAR programs passes through the supervisor (except for input of a large number of entries from a bulk storage medium). The supervisor accepts command lines (which the user may issue on a variety of input units), scans them for command names and parameters and places these latter in a list-structured buffer area.

The command language has been designed with an eye to user convenience and flexibility. In addition, the command language interfaces with the command functions in such a way as to permit relatively simple system modifications. Important features of the supervisor and command language are summarized below:

(1) The terminal user communicates to LISTAR entirely by way of simple commands which have the same format-free structure.
(2) The LISTAR command set is open-ended and indefinitely expandable through the addition of command subroutines.
(3) Since command subroutines are independent of each other, they may be grouped into system modules or segments to be executed as needed to service the user.
(4) The supervisor and command language programs are independent of the command function subroutines to permit non-terminal task programs to execute LISTAR functions directly.
(5) The terminal user searches files or relations on files by moving markers which he creates and positions during a session. He is free to create up to ten markers and may move these markers up and down a file as needed. Markers are independent of each other and may be erased individually whenever their usefulness has ended.*

Figure 1 lists a sample set of the commands in the system. The command name and identifiers required by the command are typed in upper case; parameter labels and a brief explanation of the command are typed in lower case. Where the user may choose among several alternative modes of the same command, the alternatives are placed in parentheses. For these cases, the user must select one of the alternatives. Optional parameters are set off by the paired set of symbols "<" and ">". Parameters are delimited by spaces. Parameter names more than one word long are hyphenated. A sequence of parameters of indefinite length is represented by a string containing the name of the first and last parameter separated by three dots (...).

Figure 2 is an example of a session using a number of the commands in Figure 1. For our purpose, we have chosen a file which is part of an information retrieval system called MEDLARS currently used by the National Library of Medicine at Bethesda, Maryland.

The example in Figure 2 was generated by a terminal user applying LISTAR to a file called "MESH" consisting of medical subject headings and stored on the Lincoln Laboratory time sharing facility.

SPACE MANAGEMENT AND FILE STRUCTURE

Space allocation and management

LISTAR enables users to reference data stored in a large number of files each consisting of a large number of entries. These entries are imagined to reside in a virtual file-set space of up to 2^{30} (one thousand million) bytes.

The virtual file set space is mapped into a virtual memory which is 256K bytes in size at present, but which can go as high as 16M bytes (the addressability

*The use of markers in LISTAR is patterned after the scheme developed by K. C. Knowlton for the Bell Laboratory's L^6 language. See Reference 2.

1. STOP

   'STOP' terminates listar and returns the user to cms.

2. LOAD fileset-name <fileset-type>

   'LOAD' reads the file set from the user's disk storage into virtual memory.

3. DEFINE FILE file-name (level-0-entry-name) field-name-1 field-type-1 field-length-1 ... fn-n ft-n fl-n

   'DEFINE FILE' creates a file description for the named file having the specified data fields.

4. BULK FORMAT format-name record-length records-per-entry
   begin-column-1 <to> end-column-1 field-name-1
   begin-column-n <to> end-column-n field-name-n

5. BULK INPUT (TAPn) listar-file-name
   (cms-filename cms-filetype)
   (TERMINAL)
   format-name <count>

   'BULK INPUT' causes entries to be read from the given bulk medium into a LISTAR file according to the bulk format.

6. PUT marker-name (FILE file-name)
   (RELATION relation-name <file-name>)

7. MOVE marker-name < count < field-name <zone> condition value > >

   'MOVE' moves the marker down a file, until either the number of entries given by 'COUNT' have been examined or the condition has been met.

8. FORMAT format-name file-name
   (NORM field-name-1 field-name-n)
   (TAB field-name-1 column-no-1 ... field-name-n column-no-n)

   'FORMAT' is used to specify formats for printing the contents of entries.

9. PRINT marker-name format-name

   'PRINT' causes selected data-field values (in the entry at which the marker is positioned) to be printed according to the format.

10. CHANGE marker-name field-name-1 value-1 ... field-name-n value-n

    'CHANGE' puts values into the specified fields in the entry at which the given marker is positioned.

11. DELETE ENTRY marker-name

    'DELETE ENTRY' removes from the file the entry at which the marker is positioned.

Figure 1

```
===>load meshfile
 READY 0.03

===>put m1 file mesh
 READY 0.02

===>format f1 mesh norm 'eng main hdg' tally 'tag word 1'
 READY 0.02

===>move m1 10 tally gt 4000
 READY 0.21

===>print m1 f1

 ENG MAIN HDG     : CORONARY DISEASE
 TALLY            :          4665
 TAG WORD 1       : C8.26.12
 READY 0.04

===>bulk format cd1 80 3 1 40 'eng main hdg' 81 95 'tag word 1' -
H==>  161 170 tally
 READY 0.56

===>bulk input terminal mesh cd1
1==>adult
2==>g1.4
3==>81452
 READY 0.21

===>put m1 file mesh
 READY 0.02

===>move m1 10 tally gt 4000
 READY 0.20

===>print m1 f1

 ENG MAIN HDG     : ADULT
 TALLY            :         81452
 TAG WORD 1       : G1.4
 READY 0.04

===>change m1 'tag word 1' g1.4.13
 READY 0.04

===>print m1 f1

 ENG MAIN HDG     : ADULT
 TALLY            :         81452
 TAG WORD 1       : G1.4.13
 READY 0.04

===>stop

T=1.54/9.34 17:19:36
```

Figure 2

limit of the IBM 360/67). The size of virtual memory is a CP/CMS parameter. The current version of CP/CMS maps virtual memory into a real core memory of 512K bytes.[3]

LISTAR files are list structured and processing of the files can result in essentially random access within virtual memory with an extreme paging-to-computation ratio during execution. In order to minimize this ratio, LISTAR maintains a disciplined check on the use of virtual space by way of space allocation and management algorithms.

Space allocation and management takes three forms in LISTAR. The first is "block" management which includes creation, deletion, and moving of blocks, the unit of data exchange in LISTAR between virtual memory and auxiliary storage. The second is "cell" management which concerns management of available storage within a block. The third is "entry" management, consisting of assigning and updating a virtual address to an entry. An "entry number" is a number assigned to each entry of a file at the time it is created. The number not only uniquely identifies the entry throughout its lifetime, it also specifies the current location of the entry within virtual memory space. Over its lifetime an entry may be moved from one part of the file to another so that the primary linking order may be maintained. The entry number contains within it all the information necessary to locate the entry in the file.

Block management

A file set is created by linking blocks of free space representing regions of virtual file space. The block is significant in two respects. First, it represents the unit exchange of data for LISTAR to and from auxiliary memory. LISTAR programs read blocks of data from auxiliary memory into the user's virtual memory space as required and, similarly, write blocks of data into auxiliary memory to store or update files. The block size is an integral number of 360/67 "pages" where a page is 4096 bytes. The number of pages per block has been fixed at one page for LISTAR. This number can be changed with relatively little program modification.

A new block is created whenever space is needed to enlarge a file in the file set and the current available space is insufficient to satisfy this need. A block is released when all data entries on the block have been deleted.

Every block is assigned a number ranging from 1 to $2^{18}-1$, at the time it is created. The number uniquely identifies the block. Reference to blocks residing in auxiliary storage is always made by block number. LISTAR files will, in general, extend over many blocks and will dynamically vary in extent. At any one time, however, a relatively small number of blocks will lie in a virtual memory area called the "file area". The size of the file area is determined by the amount of non-program space in virtual memory (where program space includes the space taken up by the CMS supervisor as well as LISTAR programs). For example, the file area might take up 20 pages of a 256K byte virtual memory. A table called the "virtual memory table" (VM table) is maintained in LISTAR program space which records the block number and location of those blocks currently in virtual memory. As blocks are needed, they are read into locations specified by the VM table. If necessary, a block is written out to auxiliary memory to make room for one that is more urgently needed.

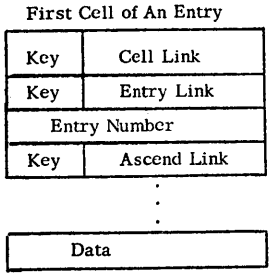
Cell management

When a block is created it is divided into smaller units called "cells". Cells may range in size from one double word to a quarter page (1024 bytes or 128 double words).

Maintenance of free space follows the scheme used by K. C. Knowlton in the Bell Laboratories' L[6] language.[4] The first word of every cell contains a key and cell link. The first bit of the key indicates whether the cell is free or in use. The next three bits identify the cell size in double words as $log_2 n$. If a cell consists of 2^n double words, then n is a three bit index on the cell size where the index may range from 0 to 7. Free space on a block is implemented as eight chains of cells, one for each cell size. The first 2-cell of every block is reserved for a free space header called Header 1. Each word of Header 1 contains a pointer to the first cell of the free cell chain of the designated size. A header word (4 bytes) is assigned to each cell size beginning with size 7 and ending with size 0. The free-space chains are doubly linked lists, that is each free cell has a pointer to its predecessor in the chain as well as to its successor. The double linking is an aid in maintaining the proper linkage as cells are acquired from and restored to the chains.

When a cell of index n is released, the corresponding twin to that cell is immediately checked to determine if it is also free. A cell is the twin of another cell if both have the same index n, are adjacent on the block and if the combined cell with index $n+1$ falls on an $n+1$ cell boundary. If the twin is also free, they are combined into a free $n+1$ cell and the process is repeated. In this fashion, the free space within a block is dynamically accumulated into the maximum size cells.

Allocation of new cells is made by first checking the



Bit		Meaning
Cell Key	0	Free/Used
	1-3	Cell size index ($\log_2$ number of double words)
	4-7	Link type
Other	0-6	Link type
	7	In-block/out-of-Block indicator

Figure 3

count for the chain of the desired size and, if it is zero, proceeding to the chain of cells of next larger size which is non-empty to decompose a larger cell. Note that Header 1 will never be selected as a free cell since its busy bit is always set.

Entry management

Entry Structure

Entries are composed of cells chained together by cell links. Since an entry will describe one object from a set of similar objects, the number and relative location within the entry of data fields which characterize the particular object are specified for the entire file by specifying the format for a typical entry of the file. This descriptive information will be described later. Although the entry format for each file is unique to that file, certain characteristics of the format are constant for all entries in the system.

Space is dedicated in the first cell of each entry for a cell key and link, a "descend" entry key and link, an entry number and an "ascend" key and link. A key is always associated with a link. A link is a pointer which always points to the first byte of a cell or entry and is either a displacement in bytes relative to the start of the block or an entry number which is convertible to a block number and byte displacement relative to the start of the block. A key is one byte long. The four high-order bits of a key in a cell link specify the cell availability and cell size as described above; for other in-block links, these bits are not used. The four low order bits of all keys specify a key value representing a link type. Five basic types are distinguished: a link to an empty list, a branch link to a sublist, a descend link to a successor cell or entry, an ascend link to a predecessor cell or entry and a return link from a sublist to its parent entry in the main list. The lowest order bit indicates whether the link is pointing to a cell or entry on the same block as itself or to an entry on another block. In this latter case, the link is called an "out-of-block" link. Figure 3 illustrates the format of the first cell of an entry in which both the entry link and ascend link point to in-block entries. If either link were an out-of-block link, it would be the entry number of the entry to which it points. LISTAR convention does not allow an entry composed of more than one cell to cross a block boundary. A cell link, therefore, will never be an out-of-block link, i.e., an entry number.

An entry in a LISTAR file may be "simple" or "complex." A simple entry consists of one or more cells linked by the cell link in decreasing cell size. The total size of a simple entry is determined initially by the space required to store the data values for the data fields specified by the user at the time the file was defined. Additional cells are acquired whenever new data fields are added and the free space internal to the entry has been exhausted. Similarly, cells are released from the entry when all the data fields assigned to the cell have been deleted. The expansion and contraction of an entry in a file is applied equally to every entry in the file.

A complex entry is a collection of simple entries which are linked together in a hierarchical structure. For convenience in exposition, we will refer to the simple entries that make up a complex entry as "sub-entries". The user determines whether his file will consist of simple entries or complex entries at the time he defines the file to the system or subsequently, if he wishes to modify the definition. He does this by giving the name of each sub-entry class, the data fields which make up each sub-entry and the parent entry to which each sub-entry is to be linked. The sub-entry class name is used by LISTAR to label the linkage between each parent entry and its list of sub-entries.

Entries on a block form a chained list which is linked on the entry link of the highest level parent. Entries are ordered in this list by the values that appear in a specially designated field called the chief field.* As entries are added to a file, they are inserted so as to retain this logical sequence. Under some circumstances this may require moving an entry to another block or creating a new block to acquire space from the free space chains. The algorithm for managing the addition and movement of entries is as follows:

(1) The current block is inspected to determine if

*The chief field is determined by the user at the time the file is defined or described to LISTAR. See earlier section.

space is available for the new entry. If space is found it is acquired and the new entry added.

(2) If space is not available and the new entry must be added to the end of the file, then a new block is created for the entry. Otherwise the predecessor block is inspected and free space is acquired for the new entry.

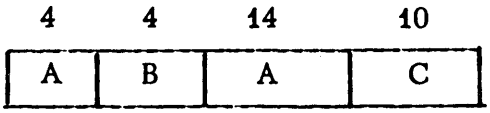
(3) If space is not available on the predecessor block, a new block is created and free space is acquired for the new entry.

(4) If the predecessor block or a new block must be used and the logical location of the new entry comes after the first entry on the current block, then the first entry is moved and the released space is used for the new entry. The entry to be moved will either be moved to the predecessor block or the newly created block.

The algorithm used by D. G. Bobrow and D. L. Murphy in their version of LISP is very similar to the one devised for LISTAR and described here.[5,6] This algorithm seeks to minimize the accessing of blocks outside virtual memory for the most frequently employed storage and retrieval operations.

Out-of-block referencing

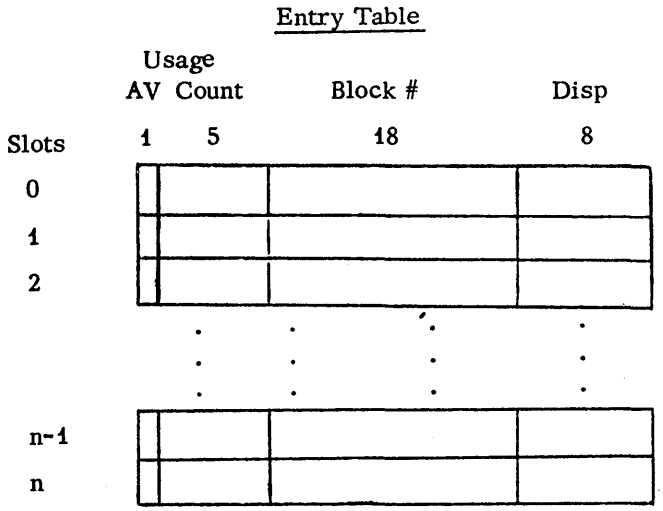
An out-of-block pointer or entry number has the following format:



where the 18 bits marked A form a block number ranging from 1 to $2^{18} - 1$; the four bits marked B represent an out-of-block key value,* the 10 bits marked C represent an index on a table called an "entry table" which resides on the block whose number is given in A.

An entry table is formed on a new block when it is created. The table serves as a directory for storing the file set location (block and displacement) of an entry at the time it is added to the file. Whenever the entry is moved, its location is updated in its table. If an entry is deleted, the deletion is noted in the entry table. By this device, the entry file set location of an entry need only be kept in one place, regardless of its actual location in the file. When an out-of-block pointer or entry number is encountered by a LISTAR routine, the entry number is decomposed into its 18 bit block number and its 10 bit index. The block number specifies the block on which the entry table for the entry resides. The index is used to determine the slot in the entry table where the file set location is stored.

The format of the entry table is shown below:



Each slot in the table is one word (4 bytes) in length. An entry table can range in size from 64 slots up to the maximum number that can fit on a block. Normally an entry table will have 64 or 128 slots. In the unusual event that all the slots of a table have been used and none are available for a new entry, a special entry table is created which fills all the space on another block exclusive of Header 1 and Header 2, i.e., 1008 slots. The block containing such a table is called a "table block". Table blocks are created as needed. In most cases a slot will be available for a new entry so that the need for a table block should be very infrequent.

An entry table is formed by acquiring a cell from the free space chains. The first bit of the cell is an availability bit (Av in the diagram) and is set to zero to indicate the cell is in use. The remaining bits of a slot are used to designate the usage count of the slot, the block number and the relative block displacement of the entry identified by the entry number. The relative block displacement is stored in units of 16 bytes (double-double words). If an entry is deleted, its entry number is retired, and its slot is marked as vacant. The slot is then available for use by a new entry. The number of times a slot may be re-used is determined by the table size. This number will be referred to as the "modulus". If s is the table size and m is the modulus then:

$$s \times 2^m = 1024 \quad \text{for } m = 2, 3, 4 \text{ or } 5$$

and

$$s \times 2^m = 1008 \quad \text{for } m = 0$$

*See above section.

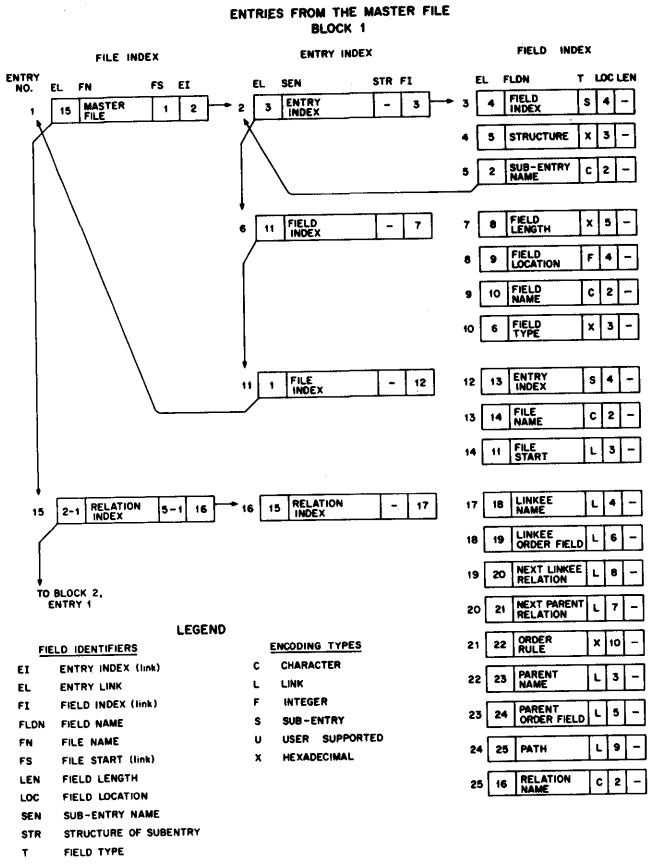


Figure 4

The usage count is recorded in the five bit field following the availability bit of the entry table. The usage count also appears as the low order m bits of the index field of the entry number, where m is the entry table modulus. Information necessary to access the entry table is recorded in a second header at the top of the block called Header 2.

FILE DESCRIPTION

There are three principal system files maintained for each file set. These are called the Master File, the Relation Index and the Path File. The Master File contains a description of every file in the file set including the Master File itself and the other system files. The descriptions of the Master File and Relation Index are always the first and second entries in the first block of a file set. Figure 4 shows a simplified illustration of a typical first block. The block has an entry which describes the Master File itself and a second entry which describes the Relation Index. We will explain the makeup of the first entry on the block (the entry which describes the Master File itself) here and explain the second entry (Relation Index) in the next section when we take up relations.

All entries in the Master File are complex entries. The Master File entry consists of three sub-entry classes called "File Index", "Entry Index" and "Field Index". The sub-entries are numbered 1 to 14 to represent entry numbers. For this illustration links are shown as entry number pointers. A number of the fields, such as key field, have been omitted for the sake of simplicity. In practice, free space might exist within some of the sub-entries and would be available for additional fields if needed. This free space is not shown in the figure.

The field location and length in the Field Index specify the position of a field relative to the start of the entry of which it is a part and its length in bytes. The numbers in the figure are illustrative only and refer to positions in the diagram. Omitted values are indicated by a dash (—). The field type specifies the internal coding format of a data value stored in the designated field. LISTAR accepts 8 types: integer, floating points, character, decimal, binary link, sub-entry and user. The "sub-entry" type identifies the linkage between a parent entry and its sub-entry list. The "user" type identifies a type defined by the user. It provides the user with a means of storing information which is coded in a form that is especially meaningful to him. Input and output of data values stored in a "user" type field are handled by an I/O program written by the user which interfaces with the LISTAR routines. The I/O program would normally be prepared by the user before defining his file.

Figure 5 illustrates a second block containing the file description of three other files. Two of the files, MESH and PIC, are data files; the third is a special file created by LISTAR to implement a relation called "ZEUS".

As indicated above MESH is a file of standardized medical subject headings. PIC is a special vocabulary of medical terms appropriate to Parkinson's disease. ZEUS is a relation which maps PIC terms to MESH terms. The relation ZEUS will be explained in the following section. The descriptive information for each of these files is stored as complex entries in the Master File. The entry link of the last entry of block 1, "Reaction Index," points to the first entry on block 2, "MESH", by way of an out-of-block pointer which is here represented by a block and entry number (2-1). The last entry of block 2 is the last entry of the Master File and points back to the first entry on block 1 (1-1).

Figure 6 illustrates the makeup of the PIC and MESH files. The PIC file contains one field for the PIC term (PT). The MESH file contains three fields: the

main heading term, Eng Main Hdg (EMH), the Tally and the classification term, Tag Word 1 (TW1). The entries are linked in alphabetical order on the entry link (EL) and each returns to its respective descriptive entry in the Master File.

RELATIONS

As indicated earlier, the user can create an association between any two files in a file set or between entries of the same file by defining a relation between them. Relations are implemented as a chain of out-of-block pointers (entry numbers) which link the entries taking part in the relation.

The entries of a relation chain are distinct from the entries in the files or file on which the relation is defined, and they are stored in a separate file called a "path file" identified by the relation name. The entries in a path file are abstracts of the entries in the main files being related. A path file links abstract entries from the parent file to a set of abstract entries from the linkee file. Abstracts are created only for those entries specifically relevant to the relation. Each entry of a path

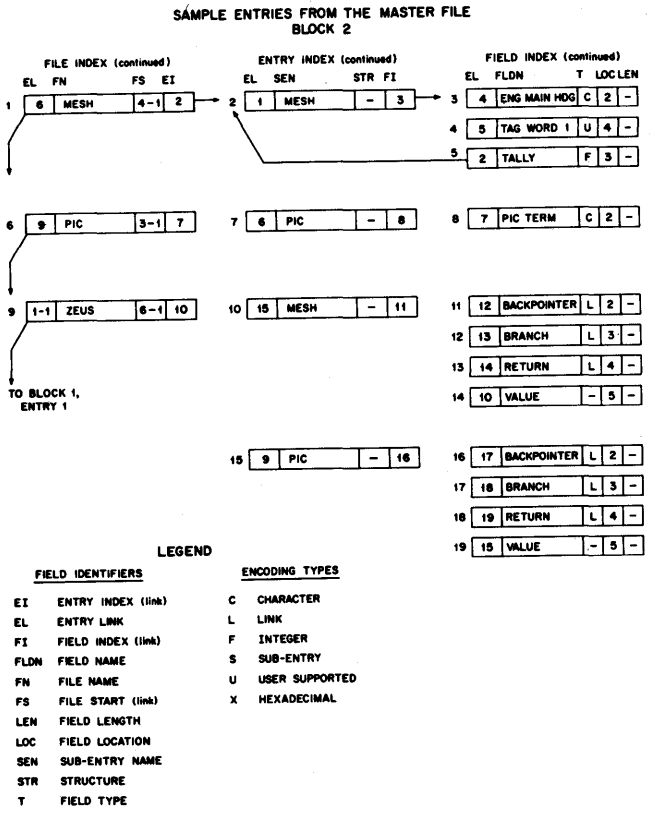


Figure 5

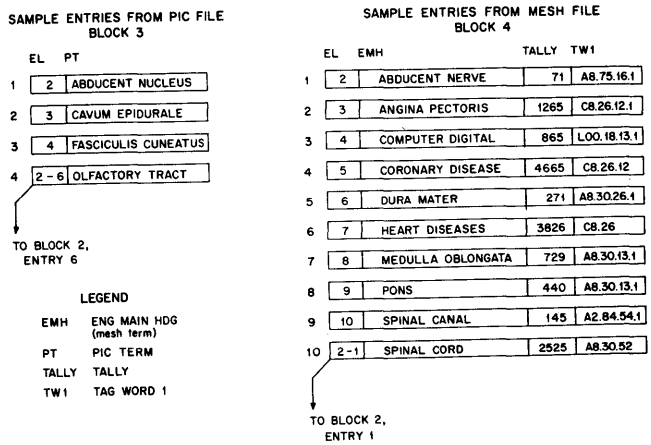


Figure 6

file, whether it be an abstract entry from the parent file or from the linkee file, has four data fields in addition to the standard fields of an entry: "back pointer", "branch link", "return link" and "value". The back pointer is the entry number of the entry from which the abstract was formed. The branch link connects a parent entry with its sub-list of linkees. The linkees are chained on their entry link. The last entry link of of the sub-list contains a key value indicating a return to its parent entry in the path file. The return link provides an additional link from the sub-list to its parent entry if one is needed for purposes of more rapid search. The value field contains a copy of the data value from each entry in the main files which takes part in the relation. The parent and linkee entries in the path file are ordered on the value field.

At the time a relation is created the user must specify the parent entries and the linkee entries that are to be associated. He must also specify the fields in the main files on which the entries in the relation file are to be ordered. The data values from each of the entries designated are copied into the value field of the relation file entries, and the entry number of the entry from which a value is copied is stored in the back pointer field of the relation file entry. The path file entries then contain just those data values which are relevant to the relation and each entry is an abstract of its correlate in the main file.

Information describing a relation is stored in the Master File and in a system file called the Relation Index. The name assigned by the user identifies a particular path file. Each such file has a description in the Master File. For example the relation ZEUS mentioned in the preceding section identifies a relation

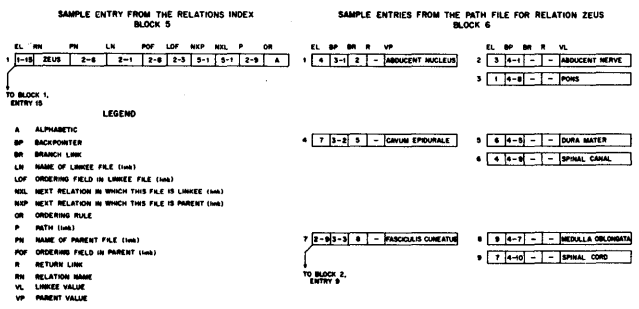


Figure 7

which maps terms from the PIC vocabulary to associated terms in the MESH vocabulary. The entry in the Master File (Figure 5) gives a description of the path file entries that take part in the relation. More specific information on each relation is stored in the Relation Index. A description of the Relation Index as it would appear in the Master File is illustrated in Figure 4. Figure 7 shows, in simple form, the entry for the relation ZEUS as it would appear in the Relation Index. The Relation Index entry contains the name of the relation (RN), the entry number of the parent file (PN), the entry number of the linkee file (LN), the entry number of the ordering field for linkee entries (LOF), the entry number of the path file description (P) and the ordering rule (OR). All values for these fields except the relation name and ordering rule are entry numbers which point to Master File entries. The ordering rule determines whether the ordering in the linkee entries is alphabetic, logical, numeric, or user defined and is specified by a preset code.

In addition to the standard chaining on the entry link, entries in the Relation Index are chained on two other links. One of these connects all Relation Index entries which have the same parent file in the relation (NXP). The other connects all entries which have the same linkee file in the relation (NXL).

When the user defines a relation he specifies the files to be related, the fields on which the parent and linkees are to be ordered, the ordering rule and the name he wishes to assign to the relation. With this information, LISTAR creates an entry in the Master File for the relation file and an entry in the Relation Index. Once the relation has been defined the user may then generate the path file by directing LISTAR to associate specific entries from the linkee file and the parent file. A particular path file may be generated incrementally by the user or, if an algorithmic association applies, by a global procedure. The user searches a relation in the same way he searches a file, by employing the commands PUT and MOVE.

The relation implementation scheme adopted for this version of LISTAR is one of several choices made possible by the LISTAR design rules. It has the disadvantage of requiring redundancy in the storage of field values, but this is offset by advantages in programming. Under this scheme the same system functions can be applied equally to the processing of relations and files with little or no need to treat relations as specialized structures.

CONCLUSION

LISTAR was designed primarily to maximize facility and flexibility in file management. Empirical evidence is insufficient at this point to determine what the cost in speed of searching might be in attaining these goals. It is, however, clear that the system is very easy to use and gives the user great latitude in creating, modifying and searching files. Ease of use has been achieved by providing a relatively simple English-like command language which requires the user to have no programming experience and very little knowledge of file organization. The self defining character of the Master File and the fact that the same structuring rules apply to the Master File as to data files make it possible to modify the Master File as easily as data files. Finally, the ability to create relations provides the user with a powerful tool for associating data entries in ways that are especially meaningful to him and which offer more rapid searching than direct searching of his primary files might allow.

REFERENCES

1 J F NOLAN A W ARMENTI
An experimental on-line data storage and retrieval system
Lincolon Laboratory MIT (TR-377) September 24 1965
2 K C KNOWLTON
A programmer's description of L[6]
Communications of the ACM Vol 9 No 7 August 1966
3 R U BAYLES ET AL
Control program—67/Cambridge monitor system (CP-67/CMS)
IBM Cambridge Scientific Center May 1 1968
4 K C KNOWLTON
A fast storage allocator
Communications of the ACM Vol 8 No 10 October 1965
5 Lincoln Laboratory Internal Technical Memorandum
October 24, 1966
6 D G BOBROW D L MURPHY
Structure of a LISP system using two level storage
Communications of the ACM Vol 10 No 3 March 1967

All-automatic processing for a large library*

by NOAH S. PRYWES

University of Pennsylvania
Philadelphia, Pennsylvania

and

BARRY LITOFSKY

Bell Telephone Laboratories
Holmdel, New Jersey

INTRODUCTION

Our concept of what is considered large-library-processing changes with the growth of published information and with the progress of the relevant data processing technology. The size of the library may be characterized by the number of entities that it concerns and the average number of retrieval *terms* that index the information about each entity. This applies to the processing of bibliographic services in preparation of recurring bibliographies of periodical literature and to the processing inherent in acquisition and custody of a library collection and communicating information regarding the collection to the library's users. In this context, a large library may be considered to have from 50,000 to tens of millions of individual publications with each publication characterized by from 10 to 100 retrieval terms. Numerous existing libraries and bibliographic services fall in this range.

Cost, personnel availability and service quality problems provide the justification for developing all-automatic processing methodology for a large library. This is similar to the justification for mechanizing any other industrial, commercial or service function. These three problem areas are particularly acute in the library field. The cost of present library processing is very high. A major component of the budget of libraries covers the processing functions which include indexing, cataloging, vocabulary maintenance, and communicating information to the library's users. There is generally a shortage of qualified personnel even where funds are available. In any one of the large libraries there are thousands of monographs and serials that are awaiting to be indexed and cataloged. These often lay unused because of the dearth of competent indexers and catalogers, especially those expert in particular subjects and languages. The increased amount of material that is being disseminated requires substantial increases in staff. Staff with such competence is extremely scarce; low salaries and monotony of processing work discourage young people from entering the library field. Finally the services are not satisfactory as indicated by the very low utilization of library resources by the scientific community. Furthermore, the libraries generally operate at a low, almost inacceptable, retrieval effectiveness and the library user requiring specific information is overwhelmed with much irrelevant information.

There are three major processing functions: (1) indexing and classifying; (2) thesaurus (vocabulary) and classification maintenance; and (3) user query interpretation. In developing an approach to the carrying out of these functions, there is a choice between the semi-automatic and all automatic processing approach. Either approach requires direct interaction of the staff and users of the library with the automatic system. However, in a *semi-automatic* approach, the staff shares with the automatic system the minute decisions in carrying out the above functions. Since such sharing of decisions and functions will continue to require expenditure in funds and highly trained staff, the semi-automatic approach will not respond fully to the problem areas delineated above. The following therefore is devoted to exploring the *all-automatic* methodology for processing in a large library.

* The research reported has received support from the Information Systems Program of the Office of Naval Research under Contract 551(40) and from the Bell Telephone Laboratories.

The methods employed in libraries for a century such as indexing and classification have proven of lasting value and serve as a foundation for the all-automatic library processing as well. Furthermore, methods of content analysis for indexing and classified concordance preparation, which require merely clerical procedures, have been proposed for centuries.[33]

To cope in a practical manner with a mass of data, the traditional approaches can be performed automatically with only minimal guidance provided by the library staff. Thus, in the all-automatic processing of a library the indexing of documents can be performed entirely by the computer following content analysis algorithms which select index terms from title, abstract, table of contents or references included in a document. This procedure also does away with the vocabulary maintenance work through maintaining an open ended index word vocabulary which is uncontrolled.

Next, an automatic process can be applied which generates a library classification system for the library collection. The classification system represents a scheme for placing documents on shelves, in microforms, in bibliographic publications, or in the computer, as appropriate. The classification system created automatically differs however from conventional classification systems employed in libraries. The prevalent approach to creating a classification system is that of applying human judgment *a priori* to divide the library collection into progressively more specific classes. By contrast, automatic classification systems are generated *a posteriori* from the information in the documents in the respective collection; namely, the index terms are extracted automatically from each document as it enters the collection. The division of the collection into progressively more specific classes is then performed automatically, based on the index terms extracted from the respective documents. The notable advantage of such a classification system is that, being generated automatically by a computer, it can always be brought to reflect an up-to-date situation incorporating new documents or using new algorithms in re-indexed documents. Thus the significance of new classes in the library collection is incorporated in the classification system.

Finally, placing publications (or bibliographic citations) according to a classification (similar to manual classification) provides the ability to browse in a library and find documents on related subjects placed together. This browsing capability can be retained in bibliographic publications (as will be shown) as well as in library shelves, in microform storage, or in the memory of the computer: wherever the documents or the surrogate information are stored. The retrieval procedure then consists of reference to a classification schedule which points to the respective areas where relevant information is found. The look up of respective index terms (or their conjunctions or disjunctions) provides the respective classification numbers, which can be algorithmically translated into storage locations. This contrasts with coordinate indexing retrieval systems, where the user's needs must be interpreted into a logical formula which should incorporate all the relevant index terms *including* appropriate synonyms. Such an interpretation requires expenditure of much time by highly skilled staff and by the computer.

The remainder of this chapter deals with three of the functions which are considered to be the major ones in an all automatic processing of a library. These are the automatic indexing which is the subject of the following section, the automatic classification which is the subject of the third section and the retrieval through browsing with the aid of a classification schedule which is the subject of the fourth section. However, the principal and novel aspect reported here is the generation and application of the Automatic-Classification in the third and fourth sections. The discussion of Automatic Indexing, in the following section, is therefore brief and of review nature.

AUTOMATIC INDEXING

A variety of automatic indexing approaches have been described by Stevens.[36]

Of interest here are only those indexing methods which do not involve human control of the indexing vocabulary. In selecting these methods there is an apparent lack of direct concern with the concepts, things, or people behind the mechanically selected index words. This is indeed not so, since to conceive anything is to represent it in symbolic form, which in the context here means representation in words which are selected.

The assimilation of new publications into the collection consists of text analysis and extraction of words through a clerical procedure. These words are then entered into a concordance of index terms. The concordance may be further processed to omit terms in some algorithmic way (based on frequency, for instance) and to form a classification system. The new publication is thus assigned a classification number and accordingly allocated storage space. Retrieval queries are similarly processed, where words in the query are extracted and used to reference the classification schedule; thereby determining the respective areas of the collection which are of interest.

The language analysis in deriving automatically the index terms may be based on the entire text, on cita-

tion and header information, such as table of contents or references, or merely on the title. The cost of transcription of the publications into machine readable form decreases greatly as the amount transcribed is reduced; however, this also reduces the eventual retrieval effectiveness. There is an indication however that effectiveness of retrieval by subject area increases considerably (20% to 25%) when the transcription of the abstract is added to the transcription of the bibliographic citation.[9,31] Content analysis of full text has not proven to sufficiently improve the effectiveness of retrieval to warrant the considerably greater cost of transcription.

Similarly, increasingly more complex language analysis procedures may be employed. However, again, the increased complexity of the algorithms may contribute only very little to the eventual retrieval effectiveness.

The simplest procedure is to analyze a text to recognize and generate stems of words encountered in the input material and treat these word-stems as candidate index terms. This involves only recognizing the suffixes of words and elimination of highly common words. Suffix editing procedures are simple; a procedure for English has been described by Stone, et al.,[37] and a procedure for French has been described by Gardin.[16] Similar procedures have been developed by numerous other investigators.[32] Such simple procedures, where stem words are derived from title or abstract, without reference to thesaurus, has proven effective for retrieval in situations where the user is satisfied with retrieval of one or few relevant documents. In a library arranged by subject, according to a classification, additional documents on relevant subjects will be found in adjacent storage areas. This simple indexing procedure can therefore serve for the all-automatic large library processing.

More sophisticated procedures employing syntactic analysis of sentences and semantic analysis involving look-up in dictionaries may be employed in the automatic indexing process. Natural language processing and machine translation research are relevant, as many of the algorithms developed there are directly applicable to automatic indexing.[30]

The aggregate of the index terms extracted from the incoming or re-indexed documents constitute an all-inclusive directory or concordance of the index terms. These directories are generated *a posteriori* from the documents themselves. An important step in deriving a usable thesaurus is the elimination of the very high and very low frequency words.[15] More sophisticated processing of the index word vocabularly may be employed. For instance, a smaller thesaurus may be obtained by including only terms with high frequency of use in retrieval queries, or index terms of documents which have been retrieved frequently. Analysis of queries may also serve as a guide regarding important relationships among terms. Statistics about frequencies of co-occurrence of terms may be used to combine terms into phrases which will be used in their entirety as a single term. Finally, the automatic generation of a classification, described later, may provide further information about grouping and sub-grouping of terms so that separate thesauri for some specific subject areas may be prepared where relationships between index words are established in the context of the subject areas.

AUTOMATIC CREATION OF A LIBRARY CLASSIFICATION

A posteriori creation of a classification

The basic aim of a classification system for a collection of documents is to group "like" documents together into categories. *A posteriori* classification does this by setting up the categories only *after* the documents are available. Thus, *a posteriori* classification, as opposed to *a priori* can optimize the categories with respect to the documents actually existing in the collection. Coupled with the automatic nature of the process, this leads to a large degree of flexibility and ability to maintain up-to-date classification schedules.

Lance and Williams[18,19] divide *a posteriori* classification strategies into *hierarchical* and *clustering* types. They further subdivide hierarchical strategies into *agglomerative* and *divisive* types. In agglomerative strategies the hierarchy is formed by combining documents, groups of documents, and groups of groups of documents until all documents are in one large group: the entire collection itself. The hierarchy being thus formed, all that remains is to select some criterion, such as category size, at which one cuts off the bottom of the hierarchy. Experiments using such a method were performed by Doyle[12,13] using the Ward grouping program.[38] Prywes[25,26,27] has also devised a system of this type, with only small scale work done on this algorithm,[39] because of computational difficulties.

Divisive techniques have long been thought the realm of philosophers and other designers of *a priori* breakdowns of knowledge. With this technique, one starts with the entire collection and successively subdivides it until appropriately sized categories are obtained. Doyle[14] has proposed a system of this type (see Dattola[11] for preliminary experiments). However, this system requires some *a priori* categories as a starting point at each level of classification. Another classification algorithm of the divisive hierarchical type is that of "CLASFY." This algorithm was devised by Lef-

TABLE I—Sample Nodes of Hierarchy

Node 1.5 *CHEMISTRY*	Node 1.5.1 *ORGANIC CHEMISTRY*	Node 1.2.3 *NUCLEAR EXPLOSIONS*
chemical reactions	organic compounds	nuclear explosion
chemical analysis	organic nitrogen compounds	radioactivity
reaction kinetics	organic sulfur compounds	radioisotopes
absorption	organic bromine compounds	contamination
stability	organic fluorine compounds	environment
solutions	methyl radicals	detection
separation process	propyl radicals	gamma radiation
uranium	isomers	temperature
impurities	amines	analysis
thermodynamics	benzene	pressure
decomposition	ethanol	computers
labelled compounds	ethers	radiation protection
thorium oxides	urea	safety
oxidation	ammonia	economics
electric potential	acetic acid	
adsorption	nitric acid	Node 1.2.3.2
lattices	heterocyclics	
cations	solvent extraction	*FISSION PRODUCTS*
spectroscopy	polymerization	
polymers	alkyl radicals	fission products
salts	oxygen compounds	filters
solubility	cycloalkanes	decontamination
organic acids	hydroxides	waste solutions
chromatography	catalysis	standards
phenyl radicals	amides	
organic chlorine compounds	hexane	
alcohols		
phenols		

kovitz.[20] Preliminary experiments have been reported[1,21] and an in-depth investigation of this algorithm along with extensive (50,000 document descriptions) testing of it has been recently (1969) performed by Litofsky.[22] Some results of these experiments will be discussed later in this chapter.

Clustering systems involve a wide variety of classification techniques which seek to group index terms or documents with high association factors together into "clusters,"[3,17,29] "clumps"[10,23,24,34,35] or "factors"[2,4,5,7,8] without trying to obtain a hierarchy. Most of these methods require matrix manipulation, though it should be added that the precise manner of these manipulations varies widely with the particular scheme used.

A tabular summary of automatic classification experiments reported upon through 1968 is presented by Litofsky.[22]

Regardless of the quality of the categories produced by a classification algorithm, the algorithm must do its task in a reasonable period of time for large collections in order to be practical for use in libraries. In most automatic classification systems being considered today (clustering types), classification time is proportional to the square, or even the cube, of the number of documents in the system. This is because of the need to compare every document (or partial category) with every other document (or partial category) or to generate and manipulate matrixes whose sides are proportional to the number of documents and/or the number of discrete keywords in the system (see Doyle,[14] "Breaking the Cost Barrier in Automatic Classification"). This means that the cost of classification *per* document goes up *at least linearly* with the number of documents. Considering collections numbering in the millions of documents, it is evident that systems with the above characteristics are unacceptable.

There are two systems which are known to break this N^2 effect (N documents in the collection). These are the two aforementioned algorithms (that described by Doyle,[14] and CLASFY) of the hierarchic, divisive type. In both, the time proportionality factor is approximately $N \log N$, where the logarithmic base is the number of branches at each node of the hierarchy. With appropriate selection of this node stratification number, the classification time (and hence cost) *per* document for these two systems can be held to a con-

stant. Using CLASFY, Litofsky[22] has estimated classification times, using third generation computers, of about .04 seconds per document, independent of the number of documents in the collection.

The resulting classification schedule

Classification schedules are required in order to be able to make use of a hierarchically classified document collection. These schedules consist of what shall be called here a "node-to-key" table (see for instance further Table I) and a "key-to-node" table (Table II). The node-to-key table is analogous to the Dewey decimal classification schedule where "node" *621.3* points to "key" *Electrical Engineering*. The key-to-node table performs the inverse function, that of producing node numbers corresponding to given keys. Because the systems under consideration here are of the *a posteriori* type, the index terms, referred to here as *keywords*, rather than *a priori* titles (such as Electrical Engineering) are present in these tables. These tables are produced by first forming a hierarchy of keywords.

The hierarchy of keywords is formed from the bottom to the top. It should be noted that this keyword hierarchy is *not* used as a semantic hierarchy in a thesaurus in order to obtain descriptors for documents, but comes about *a posteriori*. Initially, the terminal nodes, or categories, are assigned the keywords which result from the union of the keyword surrogates of the documents in that category. The keywords of the terminal nodes under a parent (next level up the hierarchy) node are then intersected and those resulting keywords are assigned to the parent node. The keyword sets of the original nodes are then deleted of the keywords assigned to the parent node. This process is continued until the top node is reached. In this way, it is guaranteed that the keywords of each document description are wholly contained in the set of keywords consisting of the keywords at the nodes in the direct path from the top of the hierarchy to the terminal node, or category, which contains that document. In addition, each keyword will appear at most once in any given path from the top of the hierarchy to a terminal node.

By this means the keywords at a node represent somewhat of an abstract of the fields of knowledge contained beneath that node. It is evident that as one goes up in the hierarchy, one will encounter more frequent or generic terms while finding the more infrequent or specific terms lower in the hierarchy. Just how this information can be used to aid browsing will be covered later.

Table I shows some sample nodes of a hierarchy generated by Litofsky[22] using CLASFY. They are part of a classification of almost 50,000 document descriptions (subject matter was nuclear science) into 265 categories. The capitalized words in Table 1 are manually assigned titles for the nodes. The node numbering scheme used is such that node 1.5.1 is directly under node 1.5 (the node stratification number equals five) and 1.2.3.2 is under 1.2.3. The node-to-key table consists of lists such as the ones shown in Table I. The key-to-node table is the inverse of this. Thus, for this example, node 1.5 would appear in the key-to-node table under "chemical reactions." Figure 1 shows a portion of the hierarchy tree (*c* indicates a terminal node or category). The keywords themselves are not shown due to lack of space.

Of course, the classification schedules are produced automatically by computer at the time the document file is classified.

TABLE II—Sample Portion of a Key-to-Node Table

MUTATIONS		*BARLEY*	
1.1.1.1.3.2.	c	1.1.1.3.1.	c
1.1.1.1.3.3.	c	1.1.1.3.2.	c
1.1.1.1.5	c	1.1.1.3.3	c
1.1.1.2.1	c	1.1.1.4.2	c
1.1.1.2.4	c	1.1.1.5	
1.1.1.3.1	c	1.1.5.1	c
1.1.1.4.1	c	1.1.5.4	c
1.1.1.4.3	c	1.3.1.4	c
1.1.1.5		1.5.3	c
1.1.3.1.1	c		
1.1.3.1.4	c		
1.1.3.5	c		
1.1.4			
1.1.5			
1.2.5.1	c		
1.5.5	c		

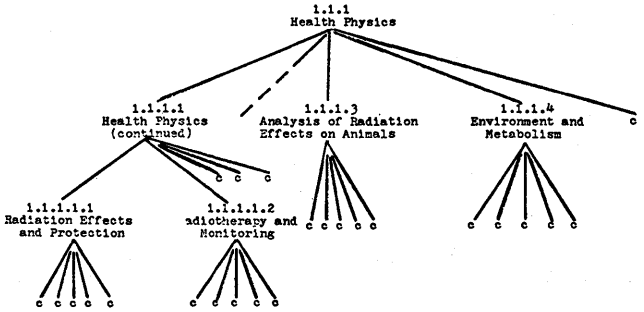


Figure 1—Portion of hierarchy

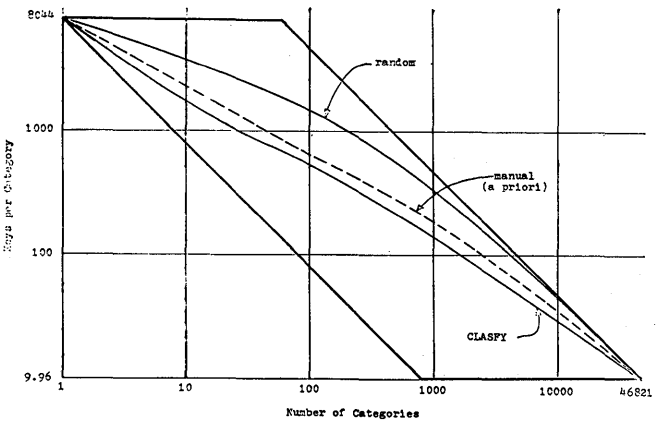


Figure 2—Keys per category, 46821 documents

Evaluation of a classification

Evaluation of a classification is not an easy matter. Until recently, almost all value judgments for classification systems were sorely hampered by dependence on human judgment. One example of this can be found by examining the automatically produced nodes of Table I. The keywords for each node seem (at least to the authors) to represent reasonably coherent areas of knowledge. However, this is a qualitative, not quantitative, measure and being thus makes it very difficult to compare this system with others.

A number of quantitative measures have been described by Litofsky.[22] Two of these will be discussed here. The "likeness" of two documents can be measured, to some extent, by the number of common index terms of these documents. Extending this notion, a measure of the quality of a classification system is how well the classification algorithm minimizes the average number of different keywords per category.

Figure 2 presents curves for this measure applied to almost 50,000 document descriptions for CLASFY, an existing manual classification system (*a priori*, documents classified by subject specialists) and for control purposes, a randomly ordered file. The parallelogram represents the theoretical boundaries for these curves.[22,28] Besides indicating closer content in categories, a lower curve also represents smaller storage requirements for the classification schedules. It is evident that CLASFY outperforms the manual systems with respect to this measure.

The second measure directly affects retrieval time for on-line retrieval systems. Most mass storage devices have two components to the time required to retrieve a record. The larger component is the time required for the read mechanism to approach the vicinity of the desired information (or vice versa). This is called the access time and is itself made up of two components, motion access and latency (usually averaging one-half revolution of the recording media). The smaller component of the retrieval time is the actual data transmission time. In general:

(1) Once the access time has been "spent," it costs relatively little more to read additional data as long as another access time is not involved.
(2) An appreciable time savings can be made by reducing the required number of memory accesses.

These points are very pertinent to manual as well as computerized on-line retrieval systems because the lack of the ability to batch queries leads to a large number of memory accesses. Automatic classification takes advantage of item (1) by grouping like documents (i.e., categories) into cells which are segments of memory (shelves, pages, tracks, cylinders, etc.) which do not require more than one memory access. Thus, it costs little extra in time to retrieve an entire cell than it would to retrieve a single document.

In addition, classification reduces the number of memory accesses required by the very fact that the documents in a given cell are close to each other in content. This "likeness" increases the probability that multiple retrievals for a given query would appear in the same cell. This in turn reduces the number of cells accessed per query and hence the number of memory accesses required.

This reduction in memory accesses can be translated into greater capacity for a system. Alternatively, it might speed operations up enough to justify slower, but less costly mass storage devices.

Thus, the second measure is the number of cells searched (accessed) for a given number of retrieval requests. Figure 3 shows the numbers of cells searched

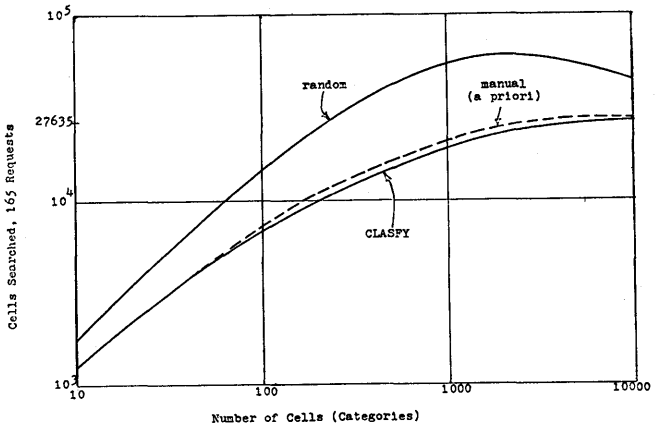


Figure 3—Cells (categories) searched, 46821 documents

in response to 165 real retrieval requests. Once again, CLASFY does better than the manual classification system.

RETRIEVAL

Browsing in the collection

Retrieval by conjunctions and disjunctions of keywords is performed by executing the proper boolean function on the node lists of the key-to-node table. This is done in a manner similar to that of inverted files with the difference being that instead of resulting in individual documents, the results here are categories.

In physical form, these categories could be books on shelves or microfilm. Once pointed toward a particular shelf or microfilm reel, the user could browse through the documents in that category to find pertinent documents.

If desired, the computer could serially scan the descriptions of the documents in a category to find the precise documents which satisfy the query.

In "The Conceptual Foundations of Information Systems," Borko[6] notes:

> "The user searches for items that are interesting, original, or stimulating. No one can find these for him; he must be able to browse through the data himself. In a library, he wanders among the shelves picking up documents that strike his fancy. An automated information system must provide similar capabilities."

The ability to browse through parts of a collection should be an essential portion of every IS&R system. There are many times when one has only a vague idea of the type of document desired. Browsing can help channel pseudo-random thoughts towards the information actually desired.

Browsing in the schedule

Effective browsing demands a hierarchical classification system in order to enable one to start with broad categories and work towards specifics. Automatic classification can produce such hierarchical sets of categories. In *a priori* systems, nodes are *given* names and index numbers. However, in *a posteriori* systems the node names are generated automatically and consist of the sets of keywords (see Table I). If a set of keywords is too large, humans or preferably automatic processes can be employed to condense the set and provide a suitable title for the node.

Naturally, *automated* browsing can only be effective in on-line computer systems through man-machine interaction. The user can enter nodes through keying conjunctions or disjunctions of keywords. The system would then display the nodes (showing the respective parts of the node-to-key table) beneath the original ones, as well as some statistics, such as how many documents there are beneath each node. When the user selects branches, the cycle repeats with the new nodes. If desired, one could backtrack up the hierarchy or jump to completely different portions. Once the user has narrowed his search, he can demand retrieval of some or all of the documents by specifying keywords and/or categories.

Another way of browsing in a classified set of documents is to start at the very bottom. Assume one has a specific query in mind and upon submitting it to the system, obtains only one document. If this is insufficient one could broaden the search by requesting the display of other documents in the category of the one retrieved. Since these documents are close in content to the original, they might also be satisfactory or their keywords might suggest ways for the user to refine his query in order to reference other nodes and retrieve other documents of interest.

None of these modes of browsing could be utilized by files with strict serial or inverted file organization.

Off-line browsing

An on-line computer is not a necessity for browsing in the schedule (though *if computers* are used, they should be on-line). The schedules could be made in book form and the browsing done by hand. The procedures would be similar to those outlined above, but somewhat slower due to page turning.

Table II represents a sample portion of a key-to-node table. It will be used to illustrate manual use of the classification schedules for a query consisting of the conjunction of MUTATIONS and BARLEY. A finger is placed at the first node entry under each of the terms. The numeric classification number under BARLEY (1.1.1.3.1) is higher than that under MUTATIONS (1.1.1.1.3.2), therefore, the listing of MUTATIONS is scanned until 1.1.1.3.1 is reached. This indicates that both keywords are contained in category 1.1.1.3.1. The scanning is continued on the list with the lower number under one's finger until 1.1.1.5 is reached. Since this node is not a category, both keywords appear in *all* categories starting with 1.1.1.5 (in this example there happen to be five such categories). Scanning is continued until 1.1.5 is encountered on the MUTATIONS list while 1.1.5.1 is found under BARLEY.

Since MUTATIONS is found in all categories under 1.1.5, the conjunction can be found in 1.1.5.1 (and 1.1.5.4). Continuing the scan does not result in any more common categories.

This process has resulted in a number of categories which would now be browsed through. The major advantage of this method is that in the indicated categories one is likely to find relevant documents which *were not* indexed by both keywords in addition to those documents which were indexed by both MUTATIONS and BARLEY.

Thus, an automated classification system can start relatively simple and grow in complexity (and cost). The classification can be done by a batched computer with retrieval done by hand from printed classification schedules. When collection size and system utilization warrants, the retrieval function could be converted to an on-line computer with little conceptual difference from the user's point of view.

Browsing simplicity

The methods of browsing outlined above are strikingly similar to those employed in traditionally organized libraries. They do not require users to be familiar with a thesaurus, its structure or with relations between terms. They do not require formulation of query formulae or to understand a computerized system processing of such formulae. In short, these methods do not require highly trained staff to interpret user queries but allow user direct browsing along traditional browsing patterns.

REFERENCES

1 T ANGELL
Automatic classification as a storage strategy for an information storage and retrieval system
Master's Thesis The Moore School of Electrical Engineering University of Pennsylvania 1966

2 G N ARNOVICK J A LILES J S WOOD
Information storage and retrieval—Analysis of the state of the art
Proceedings of the SJCC 537–61 1964

3 R E BONNER
On some clustering techniques
IBM Journal 8: 22–32 January 1964

4 H BORKO
The construction of an empirically based mathematically derived classification system
Proceedings of the SJCC 279–80 1962

5 H BORKO
Measuring the reliability of subject classification by men and machines
American Documentation 268–73 October 1964

6 H BORKO
The conceptual foundations of information systems
Systems Development Corporation Report No SP–2057 1–37 May 6 1965

7 H BORKO M BERNICK
Automatic document classification
JACM 10 151–62 April 1963

8 H BORKO M BERNICK
Automatic document classification Part II: Additional experiments
JACM 11 138–51 April 1964

9 C W CLEVERDON et al
Factors determining the performance of indexing systems Vol 1—design part 1 text
ASLIB Cranfield Research Project 1966
Also
C W CLEVERDON
Report on testing and analysis of an investigation into the comparative efficiency of indexing systems
ASLIB Cranfield Project October 1962

10 A G DALE N DALE
Some clumping experiments for associative document retrieval
American Documentation 5–9 January 1965

11 R T DATTOLA
A fast algorithm for automatic classification
In Information Storage and Retrieval Cornell University Dept of Computer Science Report No ISR–14 Section V October 1968

12 L B DOYLE
Some compromises between word grouping and document grouping
Symposium on Statistical Association Methods for Mechanized Documentation 15–24 1964

13 L B DOYLE
Is automatic classification a reasonable application of statistical analysis of text?
JACM 12 473–89 October 1965

14 L B DOYLE
Breaking the cost barrier in automatic classification
AD 636 837 July 1966

15 J S EDWARDS
Adaptive man-machine interaction in information retrieval
Ph.D. Dissertation The Moore School of Electrical Engineering
University of Pennsylvania December 1967

16 J C GARDIN
Syntol Vol II
Rutgers State University 1965

17 R T GRAUER M MESSIER
An evaluation of Rocchio's clustering algorithm
In Information Storage and Retrieval Cornell University Dept of Computer Science Report No ISR–12 Section VI June 1967

18 G N LANCE W T WILLIAMS
A general theory of classificatory sorting strategies
I Hierarchical systems
Computer Journal 9–4 373–80 February 1967

19 G N LANCE W T WILLIAMS
A general theory of classificatory sorting strategies
II Clustering systems
Computer Journal 10–3 271–7 November 1967

20 D LEFKOVITZ
File structures for on-line systems
Spartan Books March 1969 See Appendix B
21 D LEFKOVITZ T ANGELL
Experiments in automatic classification
Computer Command and Control Company Report No 85–104–6 to the Office of Naval Research Contract NOnr 4531(00) December 31 1966
22 B LITOFSKY
Utility of automatic classification systems for information storage and retrieval
Doctoral Dissertation University of Pennsylvania 1969
23 R M NEEDHAM
Application of the theory of clumps
Mechanical Translation and Computational Linguistics 8 113–27 1965
24 R M NEEDHAM K S JONES
Keywords and clumps
J. Documentation 20 5–15 March 1964
25 N S PRYWES
Browsing in an automated library through remote access
Computer Augmentation of Human Reasoning 105–30 June 1964
26 N S PRYWES
An information center for effective R&D management
Proceedings 2nd Congress on Information Systems Science 109–16 November 1964
27 N S PRYWES
Man-computer problem solving with multilist
Proceedings IEEE 54 1788–1801 December 1966
28 N S PRYWES
Structure and organization of very large data bases
Proceedings Symposium on Critical Factors in Data Management UCLA March 1968
29 J J ROCCHIO JR
Document retrieval systems optimization and evaluation
Doctoral Dissertation Division of Engineering and Applied Physics Harvard University 1966
30 N SAGER
A syntactic analyzer for natural language
Report on the String Analysis Programs Department of Linguistics
University of Pennsylvania 1–41 March 1966
31 G SALTON
Scientific report No ISR–11 and No ISR–12
In Information Storage and Retrieval Dept of Computer Science
Cornell University June 1966 and June 1967 respectively
32 G SALTON
Content analysis
Paper given at Symposium on Content Analysis University of Pennsylvania November 1967
33 W C B SAYERS
A manual of classification for librarians and bibliographers
Second edition Grafton and Company 1944
34 K S JONES D JACKSON
Current approaches to classification and clump-finding at the Cambridge Language Research Unit
Computer J 29–37 May 1967
35 K S JONES R M NEEDHAM
Automatic term classifications and retrieval
Information Storage Retrieval 4–2 91–100 June 1968
36 M E STEVENS
Automatic indexing: A state of the art report
National Bureau of Standards Monograph 91 1965
37 P S STONE et al
The general inquirer: A computer approach to content analysis
The MIT Press 1966
38 J H WARD JR M E HOOK
Application of a hierarchical grouping procedure to a problem of grouping profiles
Education and Psychological Measurement 23 69–92 1963
39 M S WOLFBERG
Determination of maximally complete subgraphs
The Moore School of Electrical Engineering University of Pennsylvania Report No 65–27 May 1965

Natural language inquiry to an open-ended data library

by GEORGE W. POTTS

Meta-Language Products, Inc.
New York, New York

INTRODUCTION

New technologies often create an effect similar to children bragging about their new Christmas toys. This "me-too-ism," as it exists in the proprietary software community, is unfortunate because it robs credibility from this emerging industry at a very crucial time. In this paper a new computer system and language, "MUSE,"* is presented with the intention not to follow this pattern.

"MUSE" could be said to belong to that family of languages called "non-procedural" in that it is not necessary to produce a sequential flow of programming logic to force output. This is a somewhat ambiguous concept in that "MUSE" *does* incorporate a capability for the user to embed "procedures." It is expected that this (among other features) will enable "MUSE" to be used as an information system for management as well as other disciplines that require easy access to and manipulation of large volumes of data.

In the development of "MUSE" an attempt has been made to refrain from producing anything of merely academic interest. It is the child of a very informal set of circumstances in the authors' experience which has consistently shown the need for an unstructured dialog between those who have non-routine problems and the computer that can contribute so much toward a solution.

BACKGROUND

Among the spectrum of new technologies four seem to be travelling convergent courses. They are:

—Time-sharing
—Natural programming languages
—Data management
—Management information systems

Each of these subjects has had a good bit of exposure in computer and business publications recently . . . much of it groping. There follow brief discourses, not explanations, on the above subjects with an attempt to offer a few preparatory insights. Following that, there is a description of "MUSE" and a few of its more interesting features that tie these subjects together.

Time-sharing

Computer time-sharing, although it provides remote access and real-time capability, is *primarily* a medium for nonroutine data processing. Here the interactive environment is the message. It is not where or when creative interaction takes place, it is *that* it take place.

A good bit of controversy exists just because of confusion on this first point. It is not surprising that the cult of time-sharing purists take issue when a remote polling capability is called "time-sharing." Polling algorithms are created to provide routine, low-level input and inquiry. The advantage of true time-sharing is *creative interaction*. The system designers who have diligently labored to develop a general purpose time-sharing capability must be complimented on their restraint when their work is confused with a reservation system or message switching.

Another point of confusion regarding time-sharing is due to a changing optimization emphasis. The entire orientation of batch processing is to push jobs through the computer as fast as possible. In time-sharing, as Dartmouth's Dr. Kemeny likes to point out, the optimization is more for the user. It is very difficult for some to accept the premise that machine time should be wasted so that optimum use can be made of the ma-

* "MUSE," an acronym for Machine-User Symbiotic Environment, is a trademark of Meta-Language Products, Inc.

chine-user pair. The motivation of some of the recent studies of comparative productivity between programming in a time-shared vs. a "batch" environment appears tenacious in the very same sense. While currently the worst case might be only slight increases in productivity, the *potential* of this type of creativity is huge using time-sharing while with batch it has about run out.

There are at least three major types of effort that lend themselves to the creative use of time-sharing:

1. Preprogrammed aids for scientists, engineers, etc.
2. The development and debugging of computer programs for *both* the batch and time-shared environments.
3. Information systems for inquiry *and* interpretation.

Notice that only the first of these has been exploited by the time-sharing vendors. The second use is probably a major objective of the large computer networks currently under construction. The last use is the reason for this paper.

Natural programming languages

English or "natural" programming languages have had a somewhat orderly development from the first halting steps, using mnemonics for binary machine codes, through FORTRAN and COBOL to the "natural" languages. It is interesting to note that BASIC, a language only slightly simpler than FORTRAN, is used by over 60% of all users of time-sharing.

Languages that had to be debugged in a batch processing environment developed quirks of form that have a remarkable staying power in the languages now being used in time-sharing. Slowly the older languages are being modified to permit interactive debugging without eliminating the compiler phase. This permits the compiled code still to be run in batch. Interpretive languages have gained popularity commensurate with the increased use of time-sharing. These are languages that are not reduced to a machine language level but rather to encoded forms that are never given machine control but are interpreted by other operations code. Interpreted code is usually slower running than compiled code and thus never had much use in batch shops. Not surprisingly, however, the first time-sharing languages interpreted are very similar to those that were formally compiled (e.g. QUIKTRAN). The advantage of both interpretive languages and interactive compilation is that they make error detection and correction a creative and involving experience.

The current step in language development appears to be the attempted elimination of all code sequences except the problem statement itself (e.g. APL).[1] With refinement the user ignores data input/output detail, compiler directives and any explicit statement of the sequential flow of logical events within the machine. Here then is the threshold of real natural language interaction. As the problem statement can be made to look more like English, so can the interactive experience be made more universal.

Data management

Data has been referred to as the fifth economic factor of production (as vital as land, labor, capital and management).* It is clear that a great deal of this data is numeric information and that manipulation of this data with computers is becoming indispensable in institutional operations.

The management of this data, or their organization for quick and easy reference, is a discipline with a checkered past. It would not be stretching a truth too far to say that data processing has followed the course determined by the development of hardware for interfacing with external data files. Fortunately for the computer industry the first technology of data referencing, sequential access, is most easily applied to routine data processing (that with the most obvious cost benefits). The other major mode of data referencing, direct access, has matured along with the growth of time-sharing. Now direct access devices are becoming available with data rates, capacities and reliability far more suited to nonroutine, unplanned data referencing.

Again, notice the subtle misapplication of this new potential due to lagging system software development. Many languages in time-sharing still reference data files sequentially, even though they are on a direct access device. This is forgivable when referencing small private data files, but intolerable if reference is attempted to very large, common data libraries (which, conveniently, also cannot be referenced with most languages in time-sharing). It would seem that a large, open-ended, data library that can be directly accessed, simultaneously, by a large number of time-sharing users would be the answer to most of the logistics problems of data management.

The only rub is that this structure, which is so well suited for non-routine data manipulation, is unsuitable for routine data processing, where throughput rates are far more critical. A simple solution, then, would be to have two data structures, one for each orientation. In most cases, routine data processing would use sequential access (except where transaction levels are small) and

* I have lost this reference.

non-routine data processing would maintain its own directly-accessed data library. The non-routine environment would accept data from the routine environment but not, in general, return any new data in the opposite direction. It is as though there were a semi-permable membrane between the two structures. This suggested solution would probably hold until far cheaper, larger and faster direct access devices are available.

Management information systems

Management information systems might be said to have a checkered present, as there seems to be no obsolescence of approach. The first attempts, on computers, to provide management with the necessary data for decision making culminated in "exception reporting." This resulted not because it was the best way of capturing interesting data from the routine processing environment, but because it was the easiest. As a consequence, exception reporting has two major drawbacks:

1. Someone must decide, a priori, how decisions are to be made and so design exception reports that contain statistics highlighting this methodology. This is a dangerous approach because it eliminates flexibility at the very point it is most needed.
2. To generate enough statistics to cover all eventualities is to generate too much paper to be conveniently read. It is a common experience to see stacks of exception reports used more as evidence of concern than as a decision-making tool.

It should be permitted the individual manager, in his own way, to have access to all that data that could affect his operations and to be able to form constructions from this data in a way best suited to the problem at hand.

The recent proliferation in the market place of data management systems must indicate an intuitive dissatisfaction with the old methodologies. However it is not plain that many of them offer a new alternative.

THE SYSTEM

A computer language has now been developed that:

—exists in a time-shared environment
—uses the English sentence form as its basis for man-machine communication
—incorporates a simple data-capturing tool and provides reference to a large common data library
—is oriented toward the creative involvement of managers and other non-technical types with the computer

```
?WHAT WERE SALES FOR THE LAST QUARTER OF THIS YEAR?
206.8 MILLION DOLLARS

?WHAT IS THE DEFINITION OF "SALES"?
SALES - THE DOLLAR VALUE OF GOODS SHIPPED TO PURCHASERS BEFORE DISCOUNTS

?HOW DO I GET SALES OF PHYSICAL UNITS?
THE DEFINITION OF SHIPMENTS IS THE NUMBER OF PHYSICAL UNITS SHIPPED TO
PURCHASERS.  THIS MIGHT HELP.

?GIVE ME SALES. SHIPMENTS AND ALSO SALES/SHIPMENTS FOR THE LAST FOUR
QUARTERS FOR CONSUMER PRODUCTS IF THEIR % OF MARKET IS GREATER THAN 25
AND THEIR CURRENT UNIT PRICE IS NOT GREATER THAN THEIR UNIT PRICE FOR
1\1\68.  TITLE THIS REPORT "SALES ANALYSIS SHEET".
```

Figure 1

This language is "MUSE." "MUSE" is not just the "MUSE" language. It is an interactive information *system* for non-routine problem solving and creative decision making.

"MUSE" is not a time-sharing system. It is a subsystem that interfaces with a general purpose time-sharing system. It is built in such a manner for transferability to different time-shared computers of different manufacturers.

"MUSE" can be easily understood in terms of five functional modules: normal interaction module; teaching module, data loading and maintenance module; meta-language processor; and report generator. Each of these modules, in turn, is constructed of submodules, many of which are used in common by the larger, functional modules. A description, with examples, of these modules follows:

Normal interaction module

The "MUSE" language, as it presents itself to the terminal user, is simple English sentences—questions, commands and declaratives. These sentences are composed individually or in paragraph form. A small sample of a dialog is shown to indicate its general interactive nature (Figure 1).

Questions are used for two main purposes: first, to recall and manipulate data; and second, to permit free form queries about "MUSE's" capabilities.

Commands, in general, are the user's control over the dialog process. This includes modifying sentences or words, listing of all or part of a dialog, inserting or removing sentences or words, updating data or definitions, creating new language elements, requesting report output, etc.

Declaratives are the user's statement of what a report should contain.

These three types of sentences may be intermixed in

SIMPLE QUALIFICATION

OPTION 1) SALES FOR 1965 FOR G.M.

OPTION 2) G.M.'S SALES FOR 1965

OPTION 3) SALES FOR G.M. FOR 1965

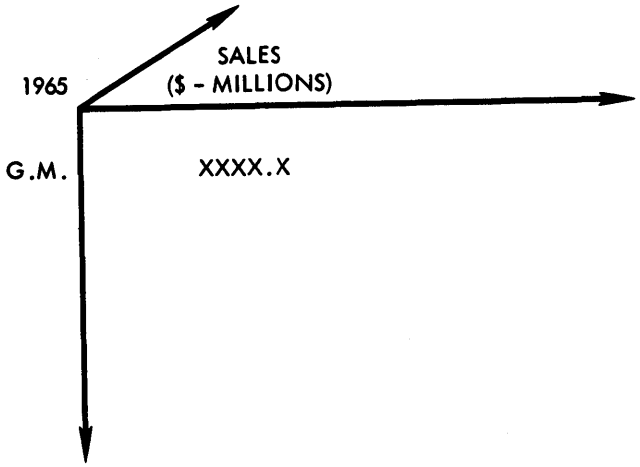


Figure 2

LISTS (SET CREATING)

SALES, EARNINGS FOR G.M., FORD FOR 1965

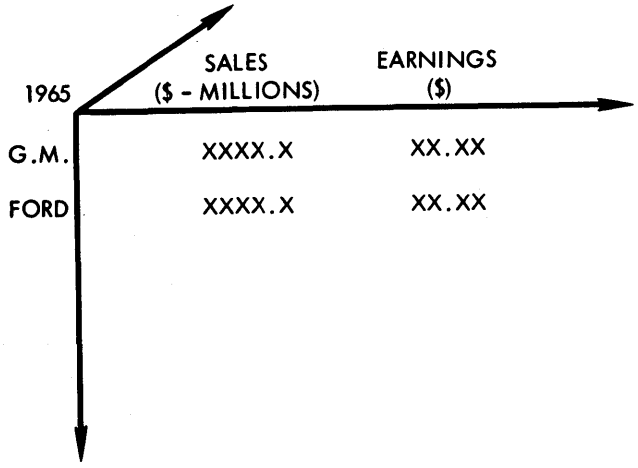


Figure 3

CLASS NAMES (UNIVERSAL SETS)

SALES FOR ALL COMPANIES FOR 1965

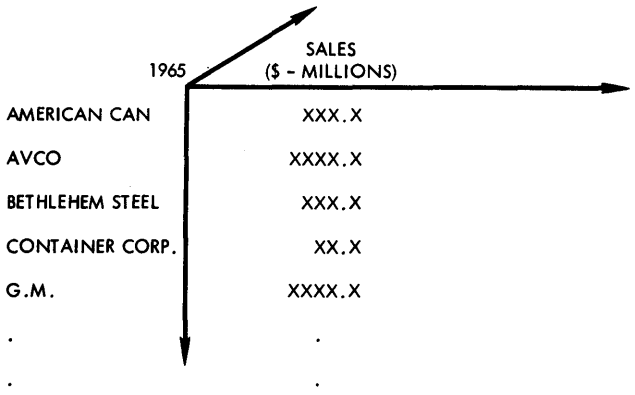


Figure 4

ARITHMETIC FORMULAE

SALES + OTHER INCOME FOR G.M., FORD FOR 1965

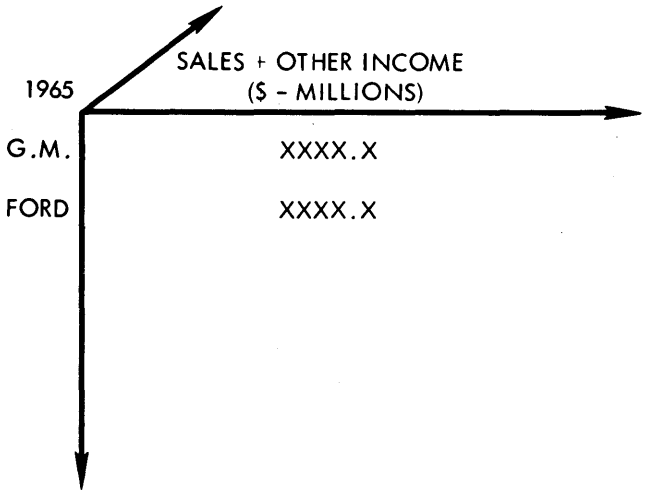


Figure 5

the dialog as the user wishes. However, declaratives are the only ones preserved when the dialog is recorded. Dialogs may also reference other dialogs and are identified in the same manner as any other entity in the system—up to 5 words of 10 alpha characters each.

In order to explain the data referencing and manipulative capabilities of "MUSE" there follows a series of eleven illustrations which represent the relationship between dialog and output. The text excerpts that produce the output may comprise part of a declarative or interrogative sentence. To visualize this, preface these excerpts with "INCLUDE" or "WHAT ARE" respectively. Also the output produced is very stylized and need not be of three dimensions.

In Figure 2, the word "for", or its synonyms is used in "MUSE" as an operator to order the qualification process. This process locates data in a data library of N dimensions by figuratively intersecting planes passing through identifier-located points on every necessary coordinate axis. The intersection of these planes produces a unique disk address for direct data reference.

In Figure 3, identifiers of like class may be explicitly grouped into lists. This normally produces a vector of data elements on output.

In Figure 4, these lists can be given identifiers and used implicitly (see Figure 8) or the identifier for a class of identifiers implies the universal set.

POINT FUNCTIONS

LOG OF SALES FOR G.M., FORD FOR 1965/1964

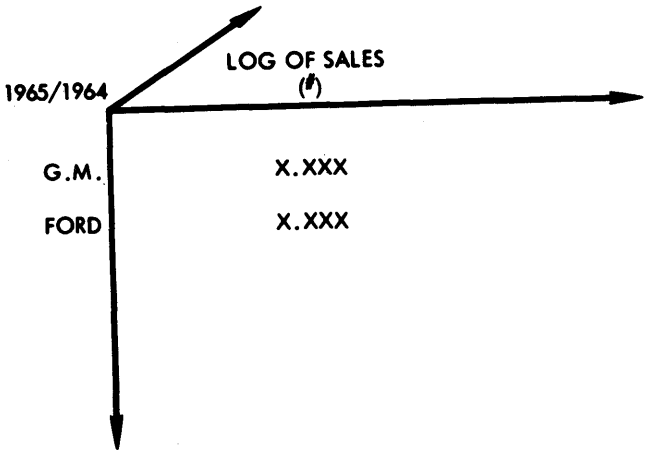


Figure 6

VECTOR FUNCTIONS

SALES, EARNINGS FOR G.M. FOR THE AVERAGE OF 1965, 1964, 1963

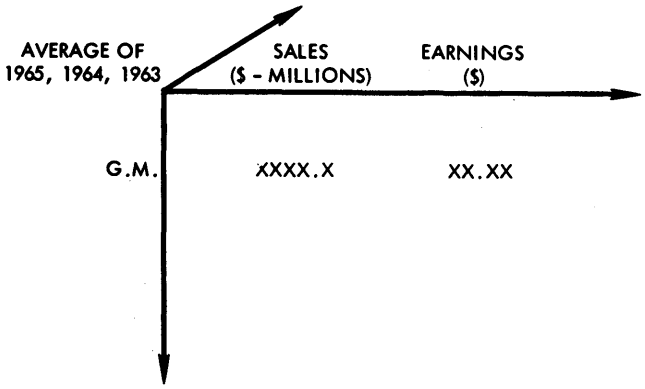


Figure 7

In Figure 5, normal arithmetic capability is available within a class of identifiers or between classes (using parentheses).

In Figure 6, functions are available to transform individual data elements on a one-for-one basis.

LANGUAGE EQUIVALENCES

1 FOR 1 (SYNONYMS)

```
EQUATE "GENERAL MOTORS INC." WITH "G.M."!
EQUATE "PLUS" WITH "+"!
EQUATE "N.F.C." WITH "NET FOR COMMON"!
```

1 FOR N (EXPRESSIONS)

```
EQUATE "AUTO INDUSTRY COMPANIES" WITH "FORD,
G.M., CHRYSLER, AMERICAN MOTORS"!
EQUATE "VALUE ADDED" WITH "SALES-COST OF GOODS
SOLD"!
```

N FOR N (PHRASE STRUCTURES)

```
EQUATE "ADD EXPRESSION TO EXPRESSION" WITH
"EXPRESSION + EXPRESSION"!
EQUATE "DEFINE EXPRESSION AS EXPRESSION" WITH
"EQUATE EXPRESSION WITH EXPRESSION"!
```

Figure 8

SET EXPRESSIONS

SALES FOR THE AUTO INDUSTRY COMPANIES WHICH ARE IN THE FORTUNE TOP 10 FOR 1965

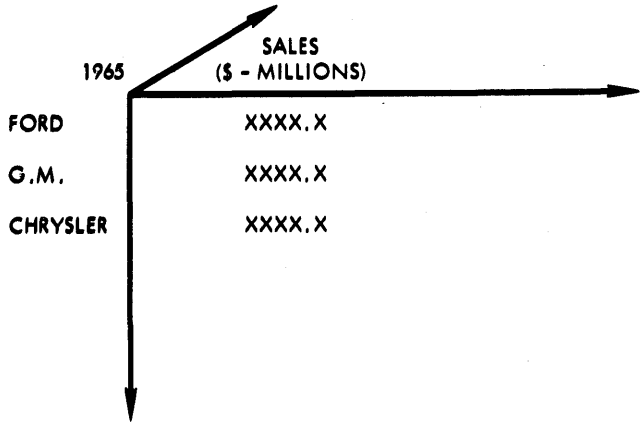


Figure 9

In Figure 7, functions are available to reduce a vector of values to a parameter.

In Figure 8, new identifiers or identifier groupings can be equated with old identifiers or identifier/operator groupings.

SIMPLE QUALIFICATION CLAUSE (WITH RELATIONAL OPERATOR)

SALES, EARNINGS FOR 1966 FOR ALL COMPANIES WHOSE SALES FOR 1965 > = $100 MILLION

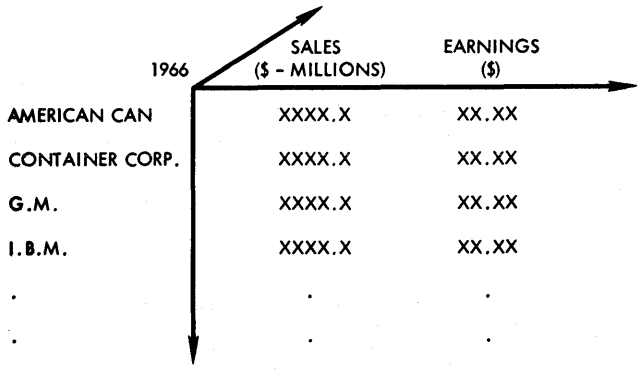


Figure 10

QUALIFICATION CLAUSE (INCLUDING BOOLEAN OPERATOR)

VALUE ADDED FOR 1966, 1967 FOR ALL COMPANIES WHOSE SALES FOR 1965 < SALES FOR 1966 AND EARNINGS FOR 1967 > $10.00

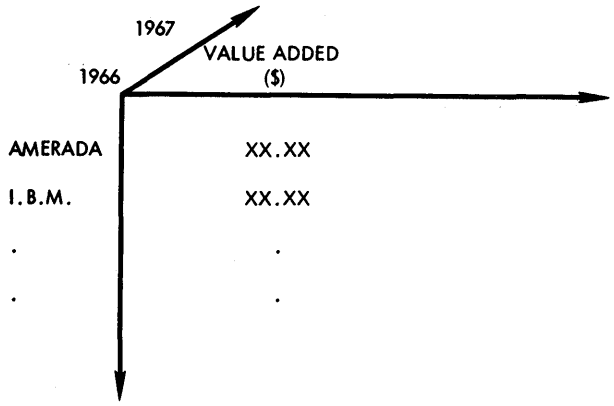


Figure 11

In Figure 9, explicit or implicit lists can be combined using standard Venn criteria.

In Figure 10, explicit or implicit lists can be culled for members which, with further qualification, meet a relational criteria.

In Figure 11, relational tests may be combined with boolean operators.

ASSIGNMENT

PROFIT FOR 1970 FOR THE PARTS DIVISION IF THE TURNOVER IS .5%

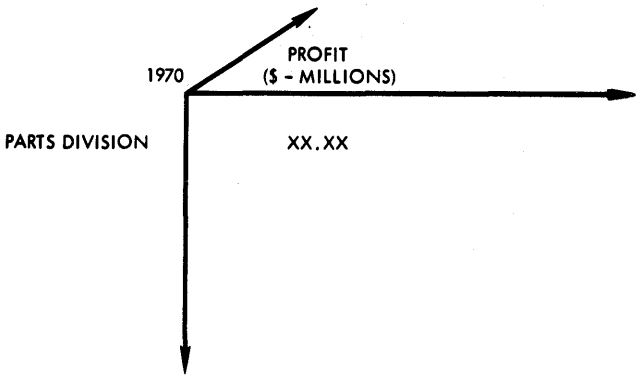


Figure 12

And in Figure 12, identifiers that exist without definitions (variables) in dialogs or language equivalences may be assigned values (symbolic or numeric) with a form of the verb "to be."

Syntax analysis in "MUSE" primarily checks operator nesting structures, juxtaposition of language tokens and parenthesis balancing.

With "MUSE," then, a satisfactory data referencing algorithm seems to be available within a sentence form that seems "natural" to its user. The latitude of possible forms is expanded by inserting language equivalences into the dictionary.

Teaching module

"MUSE" permits a question and answer interaction that bears no relation to data or calculations. This is an unstructured dialog where information is requested about how "MUSE" works. The system responds with answers that also suggest other questions. In such a manner, the user picks up the "MUSE" technique at his own speed.

This feature's first implementation is very similar in approach to that taken in the HELP system developed at Berkeley.[2] It analyzes *how*, *when* and *why* questions with a key word scheme and produces prewritten answers to anticipated questions or question sequences.

A sample of this form of dialog is provided (Figure 13).

Data-loading and maintenance module

This module could easily (and probably will) be the subject of an entire paper. The idea that there can be a generalized interface to the real world of data *input* is the dipole of the earlier concept of generalized data *output* with RPGs.

Data loading in the "MUSE" system is accomplished by first engaging in a unique question and answer dialog. Throughout this interchange the *system* is asking the questions and the user is providing the answers.

```
?HOW DO I USE MUSE?
MUSE IS USED BY TYPING QUESTIONS, COMMANDS OR DECLARATIVE SENTENCES ON
YOUR TERMINAL WHENEVER YOU SEE A QUESTION MARK (?) AT THE FAR LEFT.

?WHEN DO I USE A QUESTION?
A QUESTION IS A SENTENCE ENDING IN A QUESTION MARK (?), AS THE ONE YOU
JUST TYPED, WHICH REQUESTS IMMEDIATE INFORMATION FROM THE COMPUTER.
THERE ARE TWO TYPES OF QUESTIONS...FORMAL ONES AND INFORMAL ONES.

?HOW IS A COMMAND DIFFERENT FROM A QUESTION?
A COMMAND IS A SENTENCE ENDING IN AN EXCLAMATION POINT (!).  IT IS YOUR
CONTROL OVER THE DIRECTION OF THE DIALOG.  IT CAN BE USED TO LIST,
MODIFY OR DELETE PARTS OF DIALOGS, TO START REPORT OUTPUT, TO MODIFY
DATA VALUES OR DEFINITIONS, ETC.  SEE ABOVE FOR HOW A QUESTION IS USED.
```

Figure 13

```
2. IS A PHYSICAL INPUT REQUIRED?  YES
2.1 ON WHAT DEVICE?  TAPE
2.1.1 HOW IS THE INPUT ENCODED?  BCD
2.1.1.1 WHAT IS THE PARITY?  YES
PLEASE ANSWER FROM AMONG THE FOLLOWING: ODD, EVEN
2.1.1.1 WHAT IS THE PARITY?  EVEN
2.1.2 ARE THERE INITIAL (HEADER) RECORDS TO BE SKIPPED?  YES
2.1.2.1 HOW MANY PHYSICAL RECORDS ARE TO BE SKIPPED?  1
2.1.2.2 MAXIMUM SIZE OF SKIPPED RECORD(S)?  150
2.1.3 PHYSICAL RECORD SIZE IS?  150
2.1.4 LOGICAL RECORD SIZE IS?  300
2.2 IS PHYSICAL INPUT REQUIRED FOR DATA INPUT?  NO
2.3 IS DATA STRUCTURE (ALL/PART) DERIVED FROM INPUT?  YES
```

Figure 14

An excerpt of this form of dialog is provided (Figure 14).

These systems queries must obviously be answered by persons familiar with data processing. They will be passing on to the "MUSE" system information regarding the physical form of the data file, its logical form, the identifiers used to reference the data and other attributes of the data. This is, in effect, the bulk of the documentation normally associated with every batch processing data file.

This dialog is used to actually start the loading process and convert the data into the form used by the rest of the "MUSE" system. These dialogs are preserved for the purpose of updating with similar data files.

The only assumption made by this data-loading mechanism is that the data is in computer acceptable form and that it has been formatted for data processing (as opposed to typography, for instance).

Meta-language processor

This module performs the following major functions:

- —Parsing of character strings into language tokens
- —Syntax analysis of operator sequences and sentence forms
- —Semantic analysis using token groupings
- —Encoding to the threshold of non-reversibility into text
- —Idiomatic translation of synonyms and other more complex language equivalences
- —Polish interpretation of fully reduced dialog structures
- —External referencing of data and the resolution of other exogenous requests
- —Performance of operator requirements

"MUSE" is a proprietary product which precludes a detailed analysis of system programming techniques used to accomplish the above. Suffice it to say that it is complex in an orderly way. Its authors have taken many pains to avoid a kludged construction.

PAGE 1 OF 2 10\3\69
DIALOG - SALES ANALYSIS PRODUCED BY - MLP

SALES ANALYSIS SHEET

QUARTER 3

	SALES ($ - MILLIONS)	SHIPMENTS (FL. OZ. - THOUSANDS)	SALES/SHIPMENTS ($/FL. OZ.)
DEODORANT	15.8	921.6	.17
TOOTHPASTE	27.6	1721.3	.16
HAIR SPRAY	9.1	1010.7	.09
QUARTER 4			
DEODORANT	19.0	1021.9	.19
TOOTHPASTE	31.7	1921.8	.16
HAIR SPRAY	12.2	1232.5	.10

Figure 15

Report generator

The report reproduced (Figure 15) is part of the one developed from the declarative sentences displayed under *Normal interaction module.* Notice the symmetric structure reflecting the dimensional ordering of output. The "for" or qualifying operator has served not only to arrange the output but also to unravel the sequence of references to the data library.

The "MUSE" user, you may have noticed, has made no statement as to what is to go where in the report, how headers are to be displayed, what units are displayed, or even where the decimal point is to be placed. These were all developed, by "MUSE," from the dialog. There is *no* user intervention required to organize and produce a report from a declarative dialog. He can, however, if it be necessary, change the units and scaling of data, rotate the report axis, remove or insert extra spacing and output the report to a variety of devices.

The reason why there is such a degree of initiative on the part of the system in formatting output is that "MUSE" was designed with the assumption that it is to be used for non-routine data processing. Doing this has caused a reversal in the normal temperament of programming. Now the language is far more artistic and the output far more functional.

BUILDING BLOCK CONCEPTS

The creation of "MUSE" unequivocally depended upon the refinement of, and the belief in a small set of fundamentals. They are given here to present another lens with which to inspect the system:

Primary data

The "MUSE" system is designed to operate most effectively entirely with stored primary data. This is defined as source data, or sampling data, or data to which the construction key has been lost. It is the opposite of constructed or secondary data which are arithmetic combinations of two or more primary data elements. This does *not* mean that secondary data is not available through "MUSE." It just means that it is not stored. What are also stored are the declarative procedures that can reference primary data, combine it arithmetically, and display it *as though it had been stored.*

The advantages of this approach are as follows:

—The primary data together with declarative procedures take far less direct-access storage than the combination of primary and secondary data.
—There is no complex back indexing necessary to adjust primary data given changes in secondary data.
—There is no chance of inconsistency of result if the primary data is updated without the secondary.
—The user has far more flexibility in changing the construction of secondary data elements, or creating new ones. In fact, he may even remove entire declarative constructions without performing violence on the system.

The dimensionality of data

The realization that most numeric data, as it is organized for data processing, can assume, in this organization, a logical structure similar to a regular N-dimensional array has been fundamental in the design of "MUSE."

What has tended to obscure this point has been the great attention given to physical data structures. These physical structures become laborious due to the storing of textual information along with numeric, the great disparity between the size and number of the vector coordinates of the logical structure, and the size and dimension constraints of the physical storage medium.

The advantages of this logical form of data structure are:

—There is much less indexing information necessary to permit random retrieval. So much less that this indexing information can be stored on faster storage (e.g., drum vs. disk).
—This, of course, allows much faster accessing of data.
—It greatly simplifies the language needed for data referencing. For example, the qualification sequence "SALES FOR XEROX FOR 1965" is all that is needed to delimit a unique data element.
—It permits the direct loading and interfacing with the current data library of virtually any new data file.

Bootstrapping language

"MUSE" is an extendable language. When the "MUSE" system is installed it has a dictionary, with definitions, of approximately 250 entries. This dictionary is expanded as a result of two classes of activity:

1. The loading of data and the resultant adding of new identifiers, definitions, secondary data constructions, and identifier groupings.
2. The inserting into the dictionary of synonyms and more complex language equivalences.

The concept of a bootstrapping language is interesting because:

—The dictionary is almost entirely user-built.
—It provides the user the opportunity to communicate in his own idiom.
—The multitude of possible language forms can make the system seem very "forgiving."
—It permits both verbose *and* shorthand notation.

Information about information

In any single arithmetic computation there is really more than one thing going on. Not only are numbers being combined but also the attributes of numbers are (or should be) similarly resolved. The development of "MUSE" has taken into account this parallelism of calculation and provides the following levels of computation with every simple arithmetic operation:

—The scaling of both numbers is combined to provide the scaling of the answer.
—A count of significant digits for output is maintained.
—The units of both values are compared or combined to provide the answer units.
—Attempts are made to preserve the integer form of any numbers.

Human engineering

In the design of "MUSE" effort has been expended to make it sympatico with its user. If non-technicians are to be brought into dialog with a computer they must be appreciated for what they are, not what they might be. The adjustment must be made in the electronics and not the emotion. The following are some of "MUSE's" features that were motivated by the above realizations:

—The translation process takes place in a series of levels. Each one notes different user errors and permits correction of these errors. This is not precisely the same as incremental compilation which translates, completely, one statement at a time. In "MUSE," each statement goes through functional phases of translation which, in some cases, may be separated in time.
—"MUSE" provides full line, word, and character editing capabilities at all levels of the dialog process.
—The syntax of commands permits construction of user requests which expand on the standard VERB–OBJECT form.
—The "MUSE" system from time to time will give suggestions or warnings to the user. These are not errors, but potential errors.
—The "MUSE" system uses the Teletype character set in a standard way. "$" is meant to mean "dollars" and not some special notation to help the system designers through a rough spot.
—Definitions are maintained for every language token in the system. These definitions can be recalled at will by the user. The entire dictionary, with definitions, is also available.

Efficient systems architecture

The "MUSE" system has incorporated many efficiencies of structure:

—Modular system construction is used for across-computer-implementation and ease of upgrading capabilities.
—Assembly language coding permits economies of size and speed.
—The "MUSE" dialogs are, in fact, applications programs for the sake of reproducibility of output.
—As a subsystem, "MUSE" borrows features of the time-sharing system which condenses its size.
—Advanced system programming techniques are used throughout.

CONCLUSION

The general objective of "MUSE" has been to enhance the time-shared computer environment by providing a natural language for machine-user communication. It is designed to provide the manager with a medium of interaction with a large common data base, loaded and maintained by the "MUSE" system.

"MUSE" is capable of performing as a simulation model builder, statistical tool, data screening and sorting aid, and report organizer for *non-routine*, creative

use. It is not particularly intended to handle *routine* data processing problems requiring high-resolution data and non-symmetric, highly formatted output such as invoicing, payroll, billing, accounts payable and the like. This is why it is created in a time-shared environment where the machine-user interaction and not the material flow is the focus of optimization.

ACKNOWLEDGMENTS

Many thanks are given to David Homrighausen, Peter Kaiser, Harold Kurfehs and Susan Mazuy of Meta-Language Products, Inc. for their assistance in preparing this paper.

REFERENCES

1 K E IVERSON
A programming language
John Wiley & Sons Inc New York New York 1962
2 C S CARR
Help—An on line computer information system
Document No 40 20 30 Office of Secretary of Defense
Advanced Research Projects Agency Washington DC
January 19 1966

Computer instruction repertoire—Time for a change

by CHARLES C. CHURCH

Litton Systems, Inc.
Van Nuys, California

INTRODUCTION

A famous slogan a few years ago from a famous computer manufacturer was a single, simple word—THINK. And we in the computer profession must have taken this slogan to heart. One look at the voluminous material printed year after year on computer technology since the advent of the computer age—a quarter of a century ago—will convince anyone that a lot of thinking concerning computers has preceded us. The proceedings for this conference alone will cover two thick volumes, if we can judge from previous experience.

I hardly need mention the technological change in computer system hardware, sizes, speed, weight, cost, languages and other similar changes that have been wrought from those primitive days twenty-five years ago. But, some things have not altered, or only slightly so, like the Model T, they have remained invariant, upright, slow, inefficient and immune to the winds of progressive technological improvement. And if they did get us from point A to point B, if not in comfort, at least, we got there. I am referring to the basic form and rigid format of our instruction sets. I deliberately said our instruction sets because we have all inherited them from one original source.

With all due respect for our heritage, I think it is time to rethink our requirements for an adequate instruction repertoire.

THE PROBLEM

The symptoms of the problem are obvious:

- Programming costs are high and increasing.
- Program implementation times are long.
- Program modification is difficult and time consuming.
- Hardware systems are getting larger and larger.

It is legitimate to blame many of these difficulties on the increased complexity of problems that computers are being asked to solve. It is legitimate, but not helpful. Many of these problems could be alleviated if we modified our computer tool so that its capabilities more closely matched the job that needs to be done.

Following this logic, we did a study at Litton to roughly determine how the current machine instructions were being used. Tables I and II show some of our results.

Instructions	Percent	Total
Arithmetic	8.3	1,146
Data Moves	39.4	5,437
Logic	2.7	372
Shifts	2.3	312
Transfers	23.9	3,303
Jumps	12.0	1,655
I/O	0.7	99
Miscellaneous	10.7	1,480
		13,804

TABLE I—Instruction Occurrence by Instruction Class

Instructions	Percent	Total	Program Monitor	Navigation Program	Track Correlation Program
Arithmetic	10.5	100,206	177	12,370	87,659
Data Moves	38.4	369,645	6,035	21,228	342,382
Logic	1.4	14,064	66	3,160	10,838
Shifts	1.3	12,642	1	1,490	11,151
Transfer	26.6	256,589	4,172	18,601	233,816
Jumps	17.1	165,121	3,893	8,668	152,560
I/O	0.1	803	97	706	0
Miscellaneous	4.6	44,387	410	4,780	39,197
Total	100.0	963,457	14,851	71,003	877,603

TABLE II—Instruction Execution by Instruction Class

The high correlation between instruction occurrence and execution was interesting. Of real significance, however, was the small percentage of the repertoire devoted to accomplishing the job, as compared to the large percentage to placate the computer's whims.

In instruction occurrence we found arithmetic 8.3 percent and jumps 12 percent. What are we doing with the rest of the commands? Obviously, we need the "Data Moves" function, but do flow charts call for anything near 40 percent? And what of the transfers? My flow charts do not call for anything near 23 percent of the problem to be involved in transferring.

It becomes obvious that the majority of the programmer's required work is involved with placating the computer, while a relatively small remaining portion of the work is actually applied to the job.

What is the problem? Basically, it's that current machines have machine-oriented instructions. We must design computers that are more problem-oriented.

TWO SOLUTIONS TO THE PROBLEM

There are two major areas that must be attacked if a job-oriented computer is ever to exist:

a. Excessive Editing*
b. Fixed Length Instructions

Litton has been experimenting with solutions to these problem areas, and as a result has developed a technique of Automatic Editing for the first area, and a system of Variable Length Commands for the second. These Litton techniques are described in subsections which follow.

Excessive editing

Excessive editing largely results from two basic and common weaknesses in modern computers:

a. Inflexible word or character addressing
b. Excessive register orientation

Data fields do not align to the word or character boundary, and consequently a large portion of the program is devoted to converting input data into intermediate formats. This approach results in several undesirable conditions. The first, and most obvious, is that intermediate storage is required, and instructions are required to convert the data to intermediate storage format. These two factors combine to increase the total amount of memory required by the system. This memory expansion of course compounds the problem because the execution speed of the program decreases in direct proportion to the increased memory access required.

The strong register orientation of current computers is another stumbling block to efficient programming. To move data from one place in memory to another generally requires LOAD and STORE register commands, or worse yet, LOAD, LOAD, AND, AND, SHIFT, OR, and STORE.

This register orientation of modern computers results in roughly the same drawbacks as our current inadequate addressing. It demands excessive instructions to accomplish the job, and like the addressing problem, this results in:

a. Increased memory requirements
b. Decreased throughput speed

Now let's examine some solutions for these cumbersome, time-wasting operations.

Automatic editing

The negative effects of excessive editing can be dramatically reduced by the following computer features:

a. Address to the bit.
b. Automatic crossing of word boundaries.
c. Automatic editing for memory-to-memory operations.
d. Automatic editing for register operation.

Address to the bit and automatic crossing of word boundaries combine to provide field addressing. It is perfectly legitimate for a 33-bit field to start on bit 17 and cross a word boundary, and it is past the time that computer designers should recognize this. Field addressing negates the necessity for most intermediate storage (this is not meant to include tables, such as track stores). In addition, the majority of the LOAD, AND, OR, and SHIFT commands are no longer needed. This results in reduced memory requirements and increased execution speed.

Automatic editing for memory-to-memory operations would be obvious if it were generally recognized that memory-to-memory operations were required. Up to now memory-to-memory operations have been word or character oriented and generally slow. Memory-to-memory operations don't have to be ineffectual, but it is possible to design them that way. Addressing must be to the field. Memory move, or compare commands,

* Editing is that work required to ready a data field for use. For example, we want to add a memory field to a data register. The field starts at bit 2 and is 14 bits long. The work required to convert the field to a data format that can be added to the register is defined as editing.

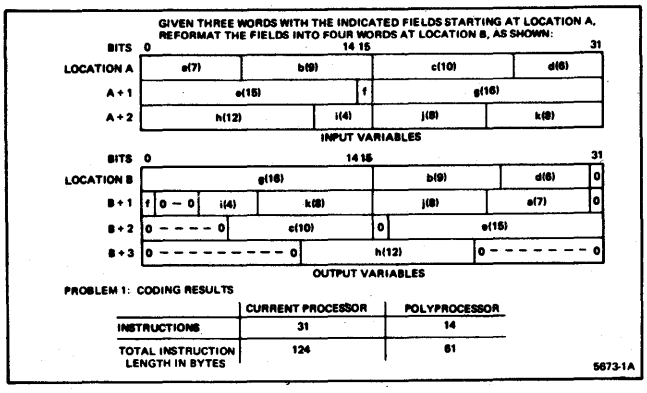


Figure 1—Problem 1: Variable length field reformatting: Format 1

must look at memory words in parallel, not some inconsequential subportion in sequence. In this way these commands can become effective and the programmer can forget the time-consuming memory juggling of the registers.

To illustrate these points we will use two reformatting problems (Figure 1 and 2). We coded these problems on a conventional processor and on Litton's new Polyprocessor. The conventional processor uses its registers for editing, while the Polyprocessor uses memory-to-memory commands with automatic editing.

Problems 1 and 2 contain exactly the same fields. The difference in the two problems is that Problem One (Figure 1) has no word boundary crossovers whereas Problem Two (Figure 2) has two boundary crossovers. The added complexity of Problem Two imposes on the conventional processor a requirement for 14 more instructions, an increase of almost 50 percent.

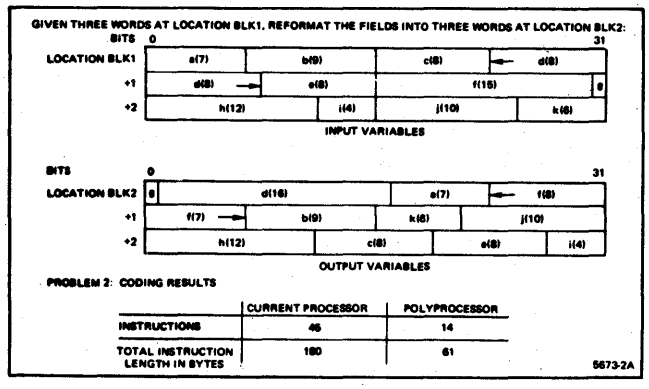


Figure 2—Problem 2: Variable length field reformatting: Format 2

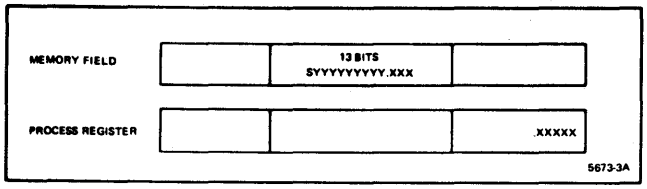


Figure 3

In contrast, the number of instructions required by the Polyprocessor remained the same, and the only operations were two extra memory accesses.

In these problems the Polyprocessor does well in the important aspects of memory required and execution speed, while the memory for the conventional processor (124 versus 61, 180 versus 61) has increased dramatically. The reason Litton's Polyprocessor executes this much faster is simply that it does not access memory as often.

The requirements for automatic editing for register operations is not as obvious now that the data formatting functions have been removed from the registers. In Litton's Polyprocessor the registers are basically used for:

a. Arithmetic computations
b. Indexing

The arithmetic area requires substantial automatic editing; however, in moving data to and from the registers the field capability removes the necessity for editing. It is now possible to say "Add the 13-bit field that starts at bit 23 to the process register."

The arithmetic process should have automatic alignment and overflow when required. For example, assume that we want to add a signed 13-bit field to a process register, Figure 3.
The ADD command will:

a. Isolate and edit the 13-bit memory field into register format, Figure 4.
b. Align the two arithmetic fields, Figure 5.

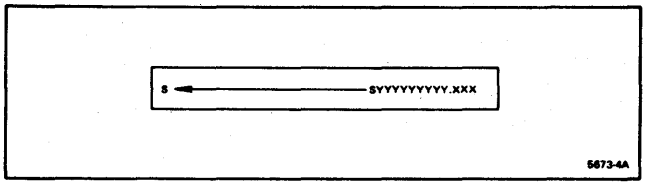


Figure 4

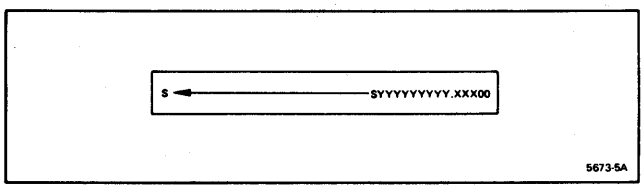


Figure 5

c. Add the two fields. If the result overflows, it will be shifted to the right one place so that it will fit.
d. Return result to the register.

In this case automatic editing for register operation has accomplished several things for us. It has removed the usually required LOAD, SHIFT, AND, Test Sign and Jump on Positive, and Set register negative commands. It has also removed the "clean up" computation commands (i.e., testing for overflow and the associated patch up commands). Finally, intermediate storage, and its associated bookkeeping, for values like this is no longer required because the value is easy to access within its input stream format.

What do all of these advantages of automatic editing mean? Very briefly we can say that they will provide you with:

a. Less programming complexity
b. Less memory required for the problem
c. Higher execution speeds because less memory is accessed

Variable length commands

The concept of fixed length commands is probably the major reason that computer instruction repertoires have made so little progress. There is no optimal command length. Timid recognition of this fact is evident in some of our modern computers, but two or three command lengths still cannot solve the problem.

For examples, let's consider the following command functions:

a. Register to register command
b. Memory to register command
c. Memory to memory command
d. Scan characters under table control.

Register to register command

This command can be accomplished with one character if the Polish Notation concept is used for register addressing. If the register were called out it would take two characters. For example add register 3 to register 5.

Memory to register command

This command requires 3 to 6 characters depending upon its implementation. For example, add location 1500 to register 5.

Memory to memory command

This command will probably require 6 to 10 characters. If we move memory to memory the command will require 2 memory addresses, a count and an operation code.

Scan characters under table control

Here we want to determine the length of the next field in a string of characters. The field length can be terminated by many characters hence a table is required to quickly identify which character will cause the scan to stop. The required instruction parameters are:

a. Data address
b. Length of character string
c. Character size
d. Table address
e. Transfer condition
f. Transfer address
g. Optional mask
h. Optional pattern address.

This command is obviously greater than ten characters in length.

The wide variation in the memory requirements for these commands demonstrate why problem-oriented instruction repertoires have not appeared. The general cry is that instructions with this range of capability will cost a fortune. Our experience at Litton says that this is not true. In fact, because smaller memories are required, we find that the total systems cost less. The cost of this type of instruction repertoire compares very favorably with common computer features, such as instruction overlap and local memories.

In summary, I can say that we at Litton are convinced that the variable length instruction is required if the computer instructions are to be problem-oriented. Further, we believe that this problem-oriented concept must be pursued, in the face of programming response time and implementation costs which have risen so high that they must be reduced. As a consequence of

	Conventional Computer	Polyprocessor	Percentage
Instruction Total	244	100	41.0
Total Memory Required in Bytes			
Actual	20,852	10,182	48.8
If Tables Optimally Compact	11,968	10,182	85.1
Microseconds	85,024.4	22,460.4	26.4

TABLE III—Program Summary

these convictions, variable length instructions have been included in Litton's new Polyprocessor.

CONCEPT VERIFICATION

To assure ourselves that we were moving in the right direction, three bench mark problems, which were previously programmed for a conventional computer, were recoded in the proposed Litton Polyprocessor instruction set. Coding for both machines was in assembly language. Also, a comparison for each problem was made as to the number and type of instructions, number of memory accesses, program memory requirements including tables, and time of execution.

The execution times for the Polyprocessor were of necessity estimated (hopefully on the conservative side) with the best information available to us at the time the study was done. The conventional computer has a 1.0-microsecond memory. The Polyprocessor has a 0.5-microsecond memory.

Our conclusion (see Tables III and IV) was that in all cases the Polyprocessor could accomplish the same result with fewer instructions and a diminished program space requirement. You will note, in particular, the complete lack of shift, logical, and move type instructions with the Polyprocessor instruction set, and the reduction in the computer "setup" instructions which are grouped under the heading of Other. The memory space saving is most apparent when comparing the space allocated to the tables.

	Conventional Computer	Polyprocessor	Percentage
Loads	77	50	64.9
Stores	11	8	72.7
Shifts	28	0	0
Arithmetics	60	39	65.0
Logical	9	0	0
Others	57	16	28.1

TABLE IV—Instruction Summary

Field	Description	No. of Bits	Parameter Range	Input or Output
X	Rectangular X Coordinate	13	±640 RM (Radar Mile = 2000 yds)	Both
Y	Rectangular Y Coordinate	13		Both
R	Polar Range	12		Both
θ	Polar Angle	12	0 to 359	Both

TABLE V—Program 1: Parameters

Bench mark program 1

There are 512 different random sets of coordinate information. Half of these sets contain polar coordinates, and the other half contain rectangular coordinates. Problem 1 is in two parts, a and b:

Problem 1a: Convert the 256 polar coordinates to rectangular coordinates using the following formulas:

$$X = R \cos \theta$$

$$Y = R \sin \theta$$

Problem 1b: Convert the 256 rectangular coordinates to polar coordinates using the following formulas:

$$R = \sqrt{X^2 + Y^2}$$

$$\theta = \arctan (X/Y)$$

A minimum end accuracy of 12 bits plus sign per coordinate position is maintained throughout the problem. Double precision arithmetic is used whenever necessary.

The parameters shown in Table V are used by both problem 1a and problem 1b.

Problem 1: coding results

Tables VI and VII display the number of instructions and other data required of each computer program to accomplish Bench Mark Program 1.

	Conventional Processor	Polyprocessor
Instructions	116	34
Memory Required in Bytes		
Instructions	464	85
Actual Constants and Tables	8,260	3,200
Optimally Compact Constants and Tables	4,164	3,200
Time (microseconds)	301	82.9

TABLE VI—Program 1: Summary

Computer	Loads	Stores	Shifts	Arithmetics	Logical	Other	Totals
Conventional Processor	37	4	15	26	6	26	114
Polyprocessor	20	4		7		3	34

TABLE VII—Program 1: Instruction Summary

	Conventional Processor	Polyprocessor
Instructions	107	46
Characters		
Instructions	428	108
Actual Constants and Tables	7,232	4,634
Optimally Compact Constants and Tables	5,100	4,634
Time (microseconds)	77,770.8	20,262.4

TABLE VIII—Program 2: Summary

Computer	Loads	Stores	Shifts	Arithmetics	Logical	Other	Totals
Conventional Processor	35	5	13	27	2	25	107
Polyprocessor	17	2	–	23	–	4	46

TABLE IX—Program 2: Instruction Summary

Field	Description	Parameter Range	Input or Output
X	Position in X	1640 RM	Input
Y	Position in Y	1640 RM	Input
FX	Battery Position in X	Actual Weapon System	Input
FY	Battery Position in Y	Position in X and Y	Input
TARG	Friendly or Foe Indicator	0 or 1	Input and Output
TIME	Time of hostile to battery	N/A	Output
VEL	Velocity of hostile	1875 yards/sec.	Constant

TABLE X—Program 3: Parameters

	Conventional Processor	Polyprocessor
Instructions	21	20
Characters		
Instructions	84	62
Actual Tables	4,384	2,093
Optimally Compact Tables	2,704	2,093
Time (microseconds)	6,952.6	2,115.1

TABLE XI—Program 3: Summary

Bench mark problem 2

There are 256 distinct tracks in the system. Half of these tracks are local and the rest are remote. Problem 2 updates each track's ground position and slant range upon each radar scan.

Each radar scan is subdivided into 20-degree sectors. All tracks within a sector are linked and are updated before tracks of another sector are considered.

Track identification for determining the actual tracks falling within a sector is provided by Track Process. This is an input array which contains a chain that links in geographical proximity.

All of the 256 track files are in the same format. Problem 2 determines the output parameter using the following formulas:

$$Xg = Xpy + XTp - T + 1/S(A_T - A_A)$$

$$Yg. \doteq Ypg + YTp - T + 1/S(A_T - A_A)$$

$$Xs = \frac{Xg^2 + 1/g^2}{Xg^2 + Yg^2 + H^2}(Xg)$$

$$Ys = \frac{Yg^2 + Yg^2}{Xg^2 + Yg^2 + H^2}(Yg)$$

Problem 2: coding results

Tables VIII and IX display data required for each computer program to accomplish Bench Mark Program 2.

Bench mark program 3

There are 256 distinct tracks in the system. Sixty-four of the tracks are hostile and are randomly dispersed among the remaining friendly 192 tracks. If a track is hostile, the designation time of the hostile to a friendly battery is computed. Table X presents the track parameters. All of the 256 track files are in the same format.

Problem 3 determines if a track is hostile. This is the case when the TARG friendly, or foe, parameter

Computer	Loads	Stores	Arithmetics	Logical	Other	Totals
Conventional Processor	5	2	7	1	6	21
Polyprocessor	6	2	7	–	5	20

TABLE XII—Program 3: Instruction Summary

is non-zero. Then the flight time of the hostile track to the battery is computed by the following formula.

$$\text{TIME} = \frac{\sqrt{(X - FX)^2 + (Y - FY)^2}}{VEL}$$

The TIME and battery position are stored in the target array. If the track is friendly, no action is to be taken.

Problem 3: coding results

Tables XI and XII display the data required of each computer program to accomplish Bench Mark Problem 3.

SUMMARY

As demonstrated by our coding experience, Litton has found that Automatic Editing and Variable Length Instructions result in:

a. Dramatic reduction in instructions required to accomplish programming jobs.
b. Dramatic reduction in memory required to hold instructions and data.
c. Substantially increased internal processing speed because fewer memory accesses are required.

Litton has incorporated these features in its Polyprocessor computer. We expect that these features, along with others, will provide users of our computer with a substantial increase in capability.

ACKNOWLEDGMENTS

The author wishes to thank Robert D. Bernstein and the staff of Data Systems Division of Litton Industries for their assistance in the preparation and editing of this paper.

The PMS and ISP descriptive systems for computer structures*

by C. GORDON BELL and ALLEN NEWELL

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

In this paper we propose two notations for describing aspects of computer systems that currently are handled by a melange of informal notations. These two notations emerged as a by-product of our efforts to produce a book on computer structures (Bell and Newell, 1970). Since we feel it is slightly unusual to present notations per se, outside of the context of particular design or analysis efforts that use them, it is appropriate to relate some background history.

The higher *levels* of computer structure—roughly, all aspects above the level of logical design—are becoming increasingly complex and, as a result, developing into serious domains of engineering design. By serious we mean the growth of techniques of analysis and synthesis, with a body of codified technique and design experience which professional designers must assimilate. In the present state, most knowledge about the technologies for computer architecture is embedded in particular studies for particular computer systems. Nothing exists comparable to the array of textbooks and systematic treatments of logical design or circuit design.

We started off simply to produce a set of readings in computer systems, motivated by this lack of systematic treatment and the inaccessibility of good examples. As we gathered material we became impressed (depressed is actually a better term) with the diversity of ways of describing these higher levels. The amount of clumsy description—downright verbosity—even in purely technical manuals acted as a further depressant. The thought of putting such a congeries of descriptions between hard covers for one person to peruse and study was almost too much to contemplate. We began to rewrite and condense many of the descriptions. As we did so, a set of common notations developed. Becoming aware of what was happening, we devoted a substantial amount of attention and effort to creating notational systems that have some consistency and, hopefully, some chance of doing the job required. These are the PMS descriptive system for what we will call the *PMS level* of computer structure (essentially the information flow level), and the ISP descriptive system for defining the *programming level* in terms of the *logic level* (actually, the *register-transfer* level).

Thus, these two notations were developed to do a descriptive task—to be able to write down the information now given in the basic machine manual in a systematic and uniform way for all current computers. They were to provide a complete defining description for complete systems, such as the IBM 7090 or the SDS 930. Hence, the essential constraints for the notations to satisfy were ones of completeness, flexibility, and brevity (i.e., high informational density).

We think the two notations meet these requirements. They have not yet been used in a way that meets additional requirements that we would all put on notational systems; namely, that there be analysis and synthesis techniques developed in terms of them.*

* This paper is taken from Chapters 1 and 2, substantially compressed and rewritten, of a book, *Computer Structures, Readings and Examples* (Bell and Newell, McGraw-Hill, 1970), which is about to be published. All figures in the paper have been reproduced with the permission of McGraw-Hill. The research in this paper was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F 44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

* There is currently a thesis in progress establishing a substantial amount of standard analysis at the PMS level. In addition, there exists at least one simulation system at the register-transfer level (Darringer, 1969) that bears a close kinship to ISP. Finally, one new computer, the DEC PDP-11, reported in this conference (Bell, et al., 1970), was designed using PMS and ISP as the working notations.

use by many people. Thus, they are undoubtedly imperfect in a number of ways (even beyond the usual questions of taste in notation, which always prevents uniform agreement and satisfaction).

By way of justification let us simply note the many places where pure descriptions (without analysis or synthesis techniques) are critical to the operation of the computer field. The programming manual serves as the ultimate educational and reference document for all programmers. Professional papers reporting on new computing systems give descriptions of the overall configuration; currently these are done by informal block diagrams. Each manufacturer adopts descriptive names of its own choosing, often for marketing purposes, to describe the components of its systems in sales brochures—e.g., selector, channel, peripheral processor, adapter, bus. During negotiations for the purchase or sale of computer system, overall descriptions (at the PMS level, as it turns out) are passed between manufacturer and potential customer. Large amounts of rough performance analyses are based on such abbreviated system descriptions. In the classroom (and elsewhere) systems are presented briefly to make particular points about design features. A user, even though he knows the order code of a machine, needs to learn the configuration available at a given installation (which, again, is a description at the PMS level). The list could be extended somewhat further, but perhaps the point is made. There is a substantial need for a uniform way of describing the upper levels of computer structures, not just for computer design, but for innumerable other purposes of marketing, use, comparison, education, etc.

With this preamble, let us describe the two notations. Notations are not theoretically neutral. That is, they are based on a particular view of the systems to be described. Thus, to understand PMS and ISP we give briefly this view of computer systems. This material is elementary and known, at least implicitly, to all computer designers. But it serves to provide the rationale for the notations and to locate them with respect to other descriptions of computer systems. After we have given some of this background, we will describe, first, PMS and then ISP. The two descriptive However, the gains to the computer field simply from the use of good descriptive notations are immense. Thus, we think that these two notations should be put forward to the computer community, both for criticism and as one concrete proposal for the adoption of a uniform notation.** The present notations are quite new and have hardly been thoroughly tested in systems have a common base of conventions, but it is simpler to treat them separately, especially when being informal. We will use the PDP-8 as an example for both PMS and ISP, since it is small enough to be completely described within the confines of this paper. At the end, in order to give some indication of generality, we will treat briefly the CDC 6600.

Our treatment here of these notations is essentially informal and heuristic. A complete treatment, as well as many examples, can be found in the forthcoming book (Bell and Newell, 1970).

HIERARCHICAL SYSTEM LEVELS

A computer system is complex in several ways. Figure 1 shows the most important. There are at least four levels that can be used in describing a computer. These are not alternative descriptions. Each level arises from abstraction of the levels below it.

A system (at any level) is characterized by a set of components, of which certain properties are posited, and a set of ways of combining components to produce systems. When formalized appropriately, the behavior

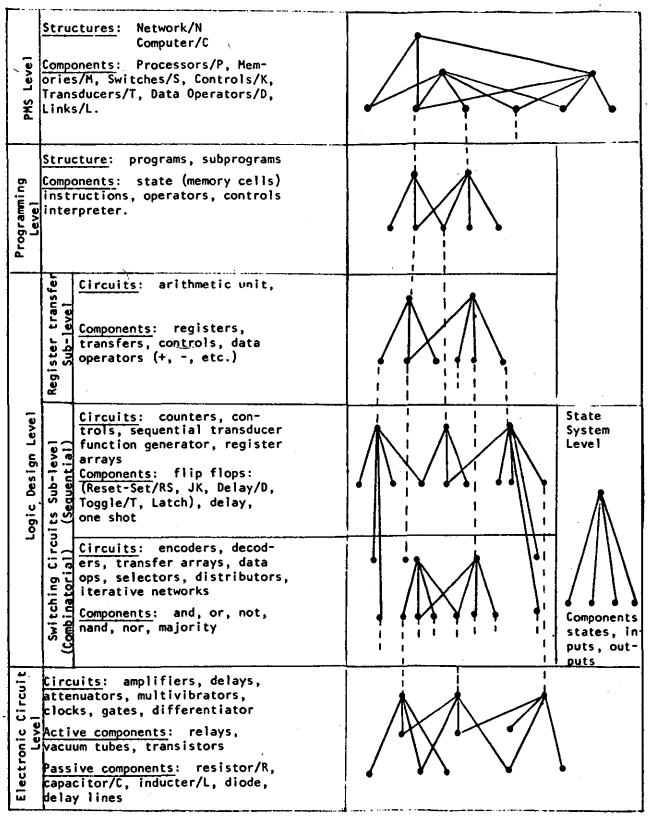


Figure 1—Hierarchy of computer structures

** A standards committee might be set up for dealing with these system levels and their description.

of the systems is determined by the behavior of its components and the specific modes of combination used. Elementary circuit theory is the prototypic example. The components are R's, L's, C's and voltage sources. The mode of combination is to run wires between the terminals of components, which corresponds to an identification of current and voltage at these terminals. The algebraic and differential equations of circuit theory provide the means whereby the behavior of a circuit can be computed from the properties of its components and the way the circuit is constructed.

There is a recursive or nested feature to most system descriptions. A system, composed of components structured in a given way, may be considered a component in the construction of yet other systems. There are primitive components whose properties are not explicable as the resultant of a system of the same type. For example, a resistor is usually not explained by a subcircuit, but is taken as a primitive. Sometimes there are no absolute primitives, it being a matter of convention what basis is taken. For example, one can build logical design systems from many different primitives (AND and NOT; NAND; OR and NOT; etc.).

A *system level,* as we have used the term in Figure 1, is characterized by a distinct language for representing and analyzing the system (that is, the components, modes of combination, and laws of behavior). These distinct languages reflect special properties of the types of components and of the way they combine. Within each level there exists a whole hierarchy of systems and subsystems. However, as long as these are all described in the same language—e.g., a subroutine hierarchy, all given in machine assembly language—they do not constitute separate system levels.

The *circuit level,* and the combinatorial switching sublevel and sequential switching sublevels of the logic level, are clearly defined in the current art. The register-transfer level is still uncertain because there is neither substantial agreement on the exact language to be used for the level, nor on the techniques of analysis and synthesis that go with it. However, there are many reasons to believe it is emerging as a distinct system level.

In the register-transfer level the system undergoes discrete operations, whereby the values of various registers are combined according to some rule, and then stored in another register (thus "transferred"). The law of combination may be almost anything, from the simple unmodified transfer ($A \leftarrow B$) to logical combination ($A \leftarrow B \wedge C$) to arithmetic ($A \leftarrow B + C$). Thus, a specification of the behavior, equivalent to the boolean equations of sequential circuits or the differential equations of the circuit level, is a set of expressions (often called productions) which give the conditions under which such transfers will be made.

There have been a number of efforts to construct formalized register transfers systems. Most of them are built around the construction of a programming system or language that permits computer simulation of systems on the RT level (e.g., Chu, 1962; Darringer, 1969). Although there is agreement on the basic components and types of operations, there is much less agreement on the representation of the laws of the system.

The *state system* representation is also at the logic level, but it has been put off to one side in Figure 1. The state system is the most general representation of discrete system available. A system is represented as capable of being in one of a set of abstract states at any instant of time. (For digital systems the set is finite or enumerable.) Its behavior is specified by a transition function that takes as arguments the current state and the current input and determines the next state (and the concomitant output). A digital computer is, in principle, representable as a state system, but the number of states is far too large to make it useful to do so. Instead, the state system becomes a useful representation in dealing with various subparts of the total machine, such as the sequential circuit that controls a magnetic tape. Here the number of states is small enough to be tractable. Thus, we have placed state systems off to one side as an auxiliary to the logic level.

The *program level* is not only a unique level of description for digital technology (as was the logic level), but it is uniquely associated with computers, namely, with those digital devices that have a central component that interprets a programming language. There are many uses of digital technology, especially in instrumentation and digital controls which do not require such an interpretation device and hence have a logic level but no program level.

The components of the program level are a set of memories and a set of operations. The memories hold data structures which represent things both inside and outside of the memory, e.g., numbers, payrolls, molecules, other data structures, etc. The operations take various data structures as inputs and produce new data structures, which again reside in memories. Thus the behavior of the system is the time pattern of data structures held in its memories. The unique feature of the program level is the representation it provides for combining components—that is, for specifying what operations are to be executed on what data structures. This is the program, which consists of

a sequence of instructions. Each instruction specifies that a given operation (or operations) be executed on specified data structures. Superimposed on this is a control structure that specifies which instruction is to be interpreted next. Normally this is done in the order in which the instructions are given, with jumps out of sequence specified by branch instructions.

In Figure 1 the top level is called the Processor-Memory-Switch level, or PMS level for short. The name is not recognized, nor is any other, since the level exists only informally. Nevertheless, its existence is hardly in doubt. It is the view one takes of a computer system when one considers only its most aggregate behavior. It then consists of central processors, core memories, tapes, discs, input/output processors, communication lines, printers, tape controllers, busses, Teletypes, scopes, etc. The system is viewed as processing a medium, information, which can be measured in bits (or digits, characters, words, etc.). Thus the components have capacities and flow rates as their operating characteristics. All details of the program are suppressed, although many gross distinctions of encoding and information type remain, depending on the analysis. Thus, one may distinguish program from data, or file space from resident monitor. One may remain concerned with the fact that input data is in alphameric and must be converted into binary, or is in bit serial and must be converted to bit parallel.

We might characterize this level as the "chemical engineering view of a digital computer," which likens it more to a continuous process petroleum distilling plant than to a place where complex FORTRAN programs are applied to matrices of data. Indeed, this system level is more nearly an abstraction from the logic level than from the program level, since it returns to a simultaneously operating flow system.

One might question whether there was a distinct systems level here. In the early days of computers almost all computer systems could be represented as in the diagram in MIT's Whirlwind Computer programming manual in Figure 2: the four classic boxes of memory (storage), control, arithmetic, and input/output (separated, in the figure). But current time-sharing and multiprocessing systems are orders of magnitude more complex than this, and it is known that the structure at this level has a determining influence on system performance. (See the PMS diagram for the 6600 in Figure 6, by no means the most complex of current systems.)

With this total view of the various systems levels we can locate both PMS and ISP. PMS is, of course, a systems level of its own, namely, the top one. ISP is a notation for describing the components and modes of combination of the programming level in terms of the

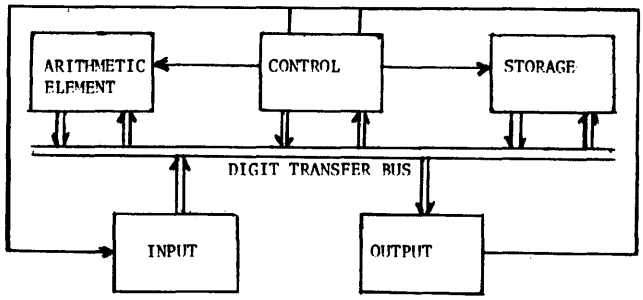


Figure 2—Simplified computer block diagram Whirlwind I (courtesy of M.I.T.)

next level down, i.e., in terms of the register transfer level. That is, the instructions, operations and interpretation cycle are the defining components of the programming level and must be given in terms of a more basic systems level. The programming level itself consists of programs written in the machine code of the system. In essence, a register-transfer description of a processor is an interpreter program for interpreting the instruction set. The interpreter describes the actual hardware of the processor. By carefully structuring a register-transfer description of a processor, instructions are precisely defined.

Thus, ISP is an interface language. Similarly, interface definitions exist at all levels of a system hierarchy, e.g., between the circuit level and the logic level. Normally, it is not necessary to have a special language for the interface; e.g., one simply writes a circuit description of an AND-gate. But with the programming level, it is most useful not to use simply a register transfer language, but to introduce a special notation (i.e., ISP). This will become clear when we describe ISP.

PMS and ISP are also strongly related in that ISP statements express the behavior of PMS components. Thus, for every PMS component there are constructs in ISP that express its behavior; and each ISP statement implies particular PMS structures.

A word should be said about antecedents. The PMS descriptive system is close to the way we all talk informally about the top level of computer systems; no one effort in the environment stands out as a predecessor. Some notations, such as CPU (for central processing units), have become widespread. We clearly have assimilated these. Our modifications, such as Pc instead of CPU, are dictated entirely by the attempt to build a consistent notation over the whole range of computer systems. With respect to ISP, we have been heavily influenced by the work on register transfer languages.* The one that we used most as a kernel

from which to grow ISP was the work of Darringer and Parnas (Darringer, 1968). In particular, their decision to work within the framework of ALGOL suited our own sensibilities, even though the final version of ISP departs from a sequential algorithmic language in a number of respects.

PMS LEVEL OF DESCRIPTION

Digital systems are normally characterized as systems that at any time exist in one of a discrete set of states, and which undergo discrete changes of state with time. Nothing is said about what physical state corresponds to a system state; or the behavior of components that transform the system from one state to another. The states are given abstract labels: S_1, S_2, ... The transitions are provided by a state-transition table (or state diagram) of the form: if the system is in state S_i and the input is I_j, then the system is transformed to S_k and evokes output O_l. The "state-system" view captures what is meant by a discrete (or digital) system. Its disadvantage is its comprehensiveness, which makes it impossible to deal with large systems because of their immense number of states (of the order 10^{10^7} states for a big computer).

Existing digital computers can be viewed as discrete state systems that are specialized in three ways. First, the state is realized by a medium, called information, which is stored in memories. Thus, a processor has all its states made explicit in a set of registers: an accumulator, an address register, an instruction register, status register, etc. No permanent information is kept in digital devices except as encoded in bits or some other information unit base in a memory. Sequential logic circuits that carry out operations in the system may have intermediate non-staticized states (e.g., during a multiply instruction), but these are only temporary. Second, the current digital computer systems consist of a small number of discrete subsystems linked together by flows of information. The natural representation of a digital computer system is as a graph which has component systems at the nodes and information flows as branches. This representation as an information flow network with functionally specialized nodes is a real specialization. Finally, each component in a digital system has associated with it a small number of discrete operations for changing its own state or the state of neighboring components. The total behavior of the system is built up from the repeated execution of the operations as the conditions for their execution become realized by the results of prior operations.

To summarize, we want a way of describing a system of an interconnected set of *components*, which are individual devices that have associated with them a set of *operations* that work on a medium of *information*, measured in bits (or some other base). For the PMS level we ignore all the fine structure of information processing and consider a system consisting of components that work on a homogeneous medium called information. Information comes in packets, called *i-units* (for information units) and is measured in bits (or equivalent units, such as characters). I-units have the sort of hierarchical structure indicated by the phrase: a record consists of 300 words; a word consists of 4 bytes; a byte consists of 8 bits. A record, then, contains $300 \times 4 \times 8 = 9600$ bits. Each of these numbers—300, 4, 8—is called a *length*.

Other than being decomposable into a hierarchy of factors, i-units have no other structure at the PMS level. They do have a *referent*—that is, a *meaning*. At the PMS level we are not concerned with what is referred to, but only with the fact the certain components transform i-units, but do not modify their meaning. These meaning-preserving operations are the most basic information processing operations of all—and provide the basic classification of computer components.

PMS primitives

There are seven basic component types, each distinguished by the kinds of operations it performs:

Memory, M. A component that holds or stores information (i.e., i-units) over time. Its operations are reading i-units out of the memory, and writing i-units into the memory. Each memory that holds more than a single i-unit has associated with it an *addressing system* by means of which particular i-units can be designated or selected. A memory can also be considered as a switch to a number of submemories. The i-units are not changed in any way by being stored in a memory.

Link, L. A component that transfers information (i.e., i-units) from one place to another in a computer system. It has fixed terminals. The operation is that of transmitting an i-unit (or a sequence of them) from the component at one terminal to the component at the other. Again, except for the change in spatial position, there is no change of any sort in the i-units.

* We have not been influenced in a direct way by the work of Iverson (Falkoff, Iverson and Sussenguth, 1964) in the sense of patterning our notation after his. Nevertheless, his creation of a full description of the IBM System/360 system in APL stands as an important milestone in moving toward formal descriptions of machines.

Control, K. A component that evokes the operations of other components in the system. All other components are taken to consist of a set of discrete operations, each of which—when evoked—accomplishes some discrete transformation of state. With the exception of a processor, P, all other components are essentially passive and require some other active agent (a K) to set them into small episodes of activity.

Switch, S. A component that constructs a link between other components. Each switch has associated with it a set of possible links, and its operations consist of setting some of these links and breaking others.

Transducer, T. A component that changes the i-unit used to encode a given meaning (i.e., a given referent). The change may involve the medium used to encode the basic bits (e.g., voltage levels to magnetic flux, or voltage levels to holes in a paper card) or it may involve the structure of the i-unit (e.g., bit-serial to bit-parallel). Note that T's are meaning preserving, but not necessarily information preserving (in number of bits), since the encodings of the (invariant) meaning need not be equally optimal.

Data-operation, D. A component that produces i-units with new meanings. It is this component that accomplishes all the data operations, e.g., arithmetic, logic, shifting, etc.

Processor, P. A component that is capable of interpreting a program in order to execute a sequence of operations. It consists of a set of operations of the types already mentioned—M, L, K, S, T and D—plus the control necessary to obtain instructions from a memory and interpret them as operations to be carried out.

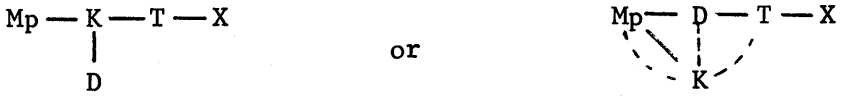
Computer model (in PMS)

Components of the seven types can be connected to make *stored program digital computers*, abbreviated by C. For instance, the classical configuration for a computer is:

$$C := Mp - Pc - T - X$$

Here Pc indicates a *central processor* and Mp a *primary memory*, namely, one which is directly accessible from a P and holds the program for it. T (input/output device) is a transducer connected to the external environment, represented by X. (The colon-equals (:=) indicates that C is the name of what follows to the right.)

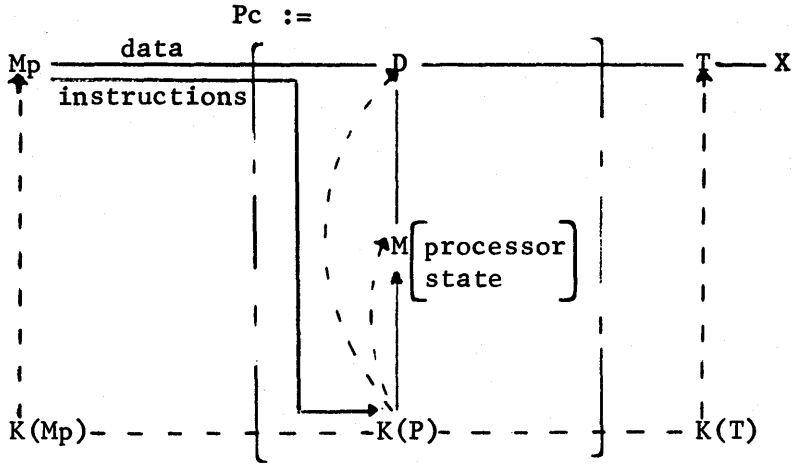
The classic diagrams had four components, since it decomposed the Pc into a control and an arithmetic unit:



where the heavy information carrying lines are for instructions and their data, and the dotted lines signify control.

Often logic operations were lumped with control, instead of with data operations—but this no longer seems to be the appropriate way to functionally decompose the system.

Now we associate local control of each component with the appropriate component to get:



where the heavy lines carry the information in which we are interested, and the dotted lines carry information about when to evoke operations on the respective components. The heavy information carrying lines between K and Mp are instructions. Now, suppressing the K's, then lumping the processor state memory, the data operators, and the control of the data, operators and processor state memory to form a central processor, we again get:

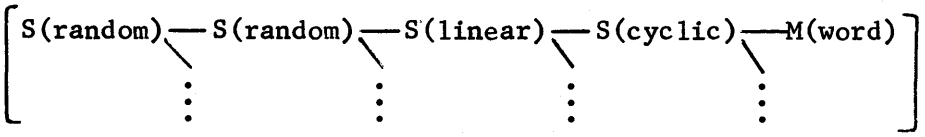
Mp—Pc—T—X

Computer systems can be described in PMS at varying levels of detail. For instance, we did not write in the links (L's) as separate components. These would be of interest only if the delays in transmission were significant to the discussion at hand, or if the i-units transmitted by the L were different from those available at its terminals. Similarly, often the encoding of information into i-units is unimportant; then there is no reason to show the T's. The same statement holds for K's—sometimes one wants to show the locus of control, say when there is one control for many components, as in a tape controller; but often this is not of interest. Then, there is no reason to show K's in a PMS diagram.

As a somewhat different case, it turns out that D's never occur in PMS diagrams of computers, since in the present design technology D's occur only as subcomponents of P's. If we were to make PMS-type diagrams of analog computers, D's would show extensively as multipliers, summers, integrators, etc. There would be few memories and variable switches. The rather large patchboard would be represented as a very elaborate manually fixed switch.

Components are themselves decomposable into other components. Thus, most memories are composed of a switch—the addressing switch—and a number of submemories. Thus a memory is recursively defined as either a memory or a switch to other memories. The decomposition stops with the unit-memory, which is one that stores only a single i-unit, hence requires no addressing. Likewise, a switch is often composed of a cascade of 1-way to n-way switches. For example, the switch that addresses a word on a multiple-headed disk might look like:

M.disk :=



The first S(random) selects a specific Ms.disk␣drive␣ unit; the second S(random) is a switch with random addressing that selects the head (the platter and side); S(linear) is a switch with linear accessing that selects the track; and S(cyclic) is a switch with cyclic addressthat finally selects the M(word) along the circular recurring track. Note that the switches are realized by differing technologies. The first two S(random)'s are generally electronic (AND-OR gates) with selection times of 10 $\sim$ 100 microseconds, or perhaps electromechanical (relay). The S(linear) is the electromechanical action of a stepping motor or a pneumatic driven arm which holds the read-write heads—the selection time for a new track is 50 $\sim$ 500 milliseconds. Finally, the S(cyclic) is determined by the rotation time of the disk and requires from 16 $\sim$ 60 milliseconds, depending on the speed (3600 $\sim$ 1000 revolutions/minute). This decomposition capability allows us to be able to describe components with varying precision and accuracy.

The control element of a computer is often shown as being associated with the processor—not to the control of a disk or magnetic tape, such a K is often more complex. When we suppress detail, controls often disappear from PMS diagrams. Alternatively, when we agglomerate primitive components (as we did above when combining Mp and K(Mp) to be just Mp) into the physically distinct sub-parts of a computer system, a separate control, K, often appears. The functionally and physically separate control* has evolved in the last decade. These controls, often larger than a Pc, are sometimes computers with stored control programs. When we decompose such a control there are: data operations (D) for calculating addresses or for error detection and error correction data; transducers (T) for changing logic signal levels and information flow widths; memory (M) as it is used in D, T, K, and for buffering; and finally a large control (K) which coordinates the activities of all the other primitives.

* A variety of names for K's are used, e.g., controller, adapter, selector, interface, buffer multiplexor, etc. Often these names reflect other functions performed by the device.

The components are named according to the function they perform and they can be composed of many different types of components. Thus, a control (K) must have memory (M) as a subcomponent, and a memory, M, may have a transducer (T) as well as a switch (S) as subcomponents. All of these subcomponents, of course, exist to accomplish the total function of the component, and do not make the component also some other type. For instance, the M that does a transduction (T) from voltages on its input wires to magnetism in its cores and a second transduction from magnetism to voltages on its output wires does not thereby become a transducer as far as the total system functioning is concerned. To the rest of the system all the M can do is to remember i-units, accepting and delivering them in the same form (voltages). We define for each component type both a simple component and a compound component, reflecting in part the fact that complex subsystems can be put together to perform a single function from the viewpoint of the total system. For example, a typewriter may have 4 $\sim$ 6 simple information transduction channels using video, tactile, auditory, and paper information carriers.

PMS notation

Various notational conventions designate specifications for a component, e.g., Mp for a functional classification, and S(cyclic) for a type of switch access function in the case of rotating memory devices like drums. There are many other additional specifications one wants to give. A single general way of providing additional specifications is used so that if X is a component, we can write:

$$X(a_1: v_1; a_2: v_2; \ldots.)$$

to indicate that X is further specified by attribute a_1 having value v_1, attribute a_2 having value v_2, etc. Each *parameter* (as we call the pair a_i v_i is well defined independently of what other parameters are given; hence, there is no significance to the order in which they are written, or to the number which have to be written.

According to this notation we should have written M(function:primary) or S(access-function:random) rather than Mp or S(random). There are conventions for abbreviating and abstracting parameters to avoid such a lengthy description. Alternative ways of writing Mp are:

M(function:primary)	complete specification
M(primary)	drop the attribute, function, since it can be inferred from the value
M.primary	use the value outside the parenthesis, concatenated with a dot
M.p	use an explicitly given abbreviation, namely, primary/p (only if it is not ambiguous)
Mp	drop the concatenation marker (the dot), if it is not needed to recover the two parts (all components are given by a single capital letter—here M)

Each of these rules corresponds to a natural tendency to abbreviate when redundant information is given; each has as its condition that recovery must be possible.

In the full description (Bell and Newell, 1970) each component is defined and given a large number of parameters, i.e., attributes with their domain of values. Throughout, the slash (/) is used to introduce abbreviations and aliases as we go.* Any list of parameters does not exhaust those aspects of a component that one might ever conceivably want to talk about. For instance, there are many quite distinct dimensions for any component in addition to the information dimension: packaging, physical size, physical location, energy use, cost, weight, style and color, reliability, maintainability, etc. Furthermore, each of these dimensions includes an entire set of parameters, just as the information dimension breaks out into the set of parameters illustrated in the figures. Thus the descriptive system is an open one and new parameters are definable at any occasion.

The very large number of parameters provides one of the major challenges to creating a viable scheme to describe computer systems. We have responded to this in part by providing automatic ways in which one can compress the descriptions by appropriate abbreviation —while still avoiding a highly cryptic encoding of each separate aspect. Abstraction is another major area in which some conventions can help to handle the large numbers of parameters. For instance, one attribute of a processor is the time taken by its operations. This attribute can be defined with a complex value:

Pc(operation-times: add:4 μs, store:4 μs, load:4 μs, multiply:16 μs, ...)

That is, the value is a list of times for each separate operation. One might also give only the range of these numbers; this is done by indicating that the value is a range:

Pc(operation-time: 4 $\sim$ 16 μs).

* There is no difficulty distinguishing this use from the use of slash as division sign—the latter takes priority, since it is the more specific use of the slash.

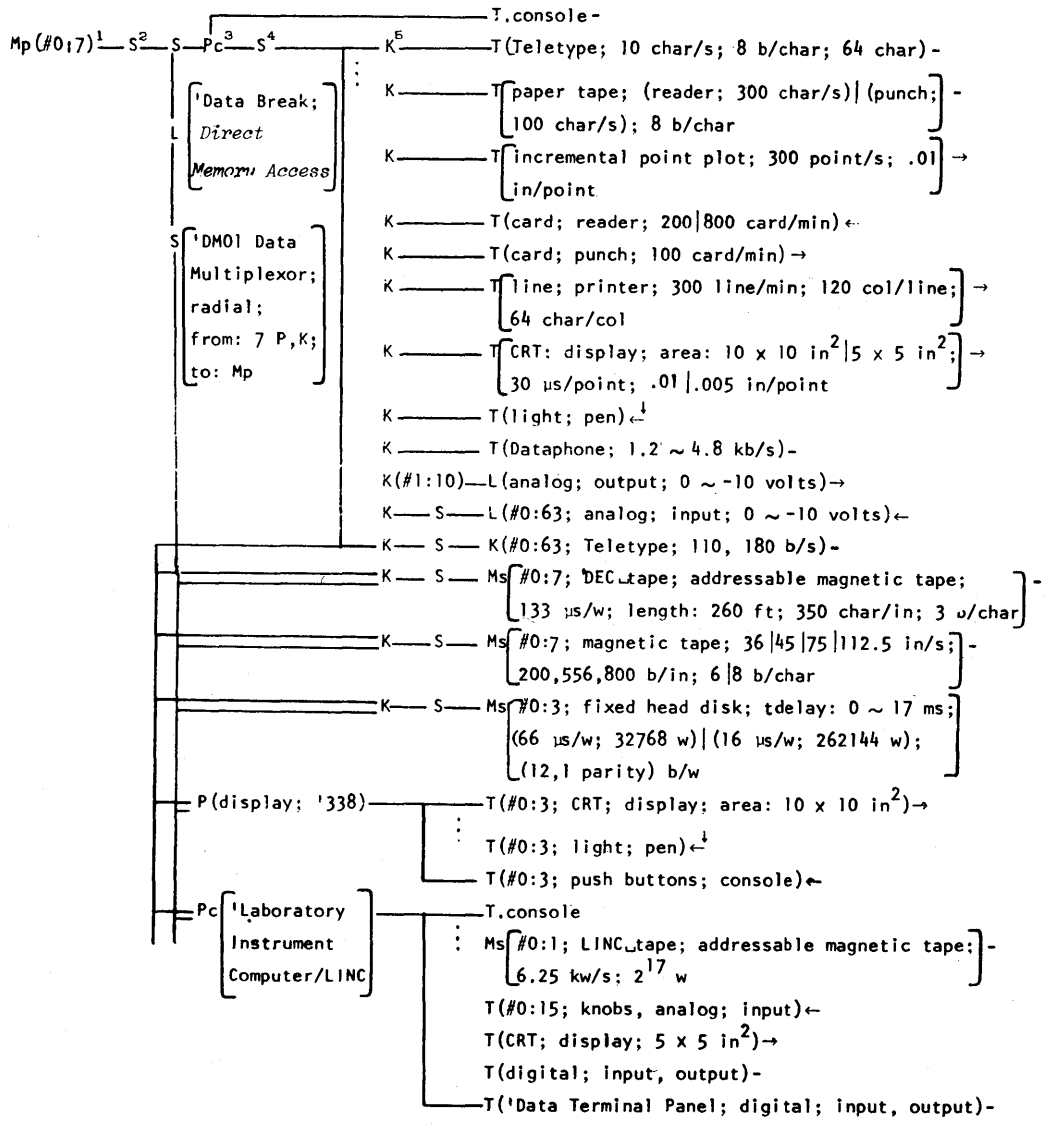
Similarly, one could have given typical and average times (under some assumed frequency mix of instructions):

Pc(operation-times: 4 μs)
Pc(operation-times: average: 8.1 μs).

The advantage of this convention, which permits descriptions of values to be used in place of actual values whenever desired, is that it keeps the number of attributes that have to be defined small.

A PMS example using the DEC PDP-8

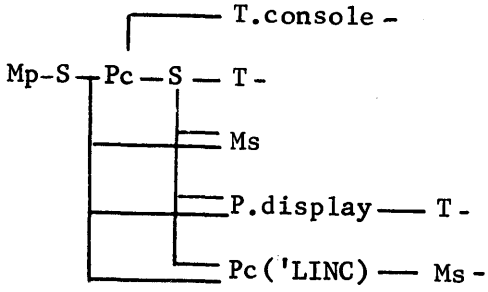
Figure 3 gives the detailed PMS diagram of an actual, small, general purpose computer, the DEC



[1]Mp(core; 1.5 μs/w; 4096 w; (12 + 1)b)
[2]S('Memory Bus)
[3]Pc(1 ~ 2 w/instruction: data: w, i,bv; 12 b/w; M.processor state($2\frac{1}{6}$ ~ $3\frac{1}{2}$) w; technology: transistors; antecedents: PDP-5; descendants; PDP-8S, PDP-8I, PDP-L)
[4]S('I/O Bus; from; Pc; to; 64 K)
[5]K(1 ~ 4 instructions; M.buffer(1 char~2 w))

Figure 3—DEC LINC-8-338 PMS diagram

LINC-8-338, which is a PDP-8 with a LINC processor and a type 338 display processor. We will concentrate on the notation, rather than discussing substantive features of the system. A simplified PMS diagram of the system shows its essential structure:



This shows the basic Mp-Pc-T structure of a C with the addition of secondary memory (Ms) and two processors, one of which, Pc('LINC), has its own Ms. Two switches are used: the I/O-bus which permits access to all the devices, and a direct access path to Mp via Pc for high data rate devices. There are many other switches in the actual system as one can see from Figure 3; for example, Mp is really 1 to 8 separate modules connected by a switch S to Pc. Also there are many T's connected to the input-output switch, S, which we collapsed as a single compound T; and similarly for S(direct memory access).

Consider the Mp module. The specifications assert that it is made with core technology, that its word size is 13 bits (12 data bits plus one other with a different function); that its size is 4096 words; and that its operation time is 1.5 μs. We could have written the same information as:

M(function:primary; technology:core; operation-time: 1.5 μs; size: 4096 w; word: (12 + 1) b)

In Figure 3 we wrote only the values, suppressing the attributes, since moderate familiarity with memories permits an immediate inference about what attributes are involved. As another example, we did not specify the function of the additional bit in the word when we wrote (12 + 1) b. Informed readers will assume this to be a parity bit, since this is the common reason for having an extra bit in a word. If the extra bit had some unusual function, then we would have needed to define it. That is, in the absence of additional information, the most common interpretation is to be assumed.

In fact, we could have been even more cryptic and still communicated with most readers:

M.core (1.5 μs/w; 4 kw; 12 b),

corresponding to the phrase, "A 12 bit, 1.5 μs, 4k core store". 4 kw stands for 4 $\times$ 1024 = 4096; however, if someone who was less familiar took it to be 4 $\times$ 1000 = 4000 no real harm would be done.

Consider the magnetic tapes for Pc. Since there are eight possible tapes that make use of the same controller, K, through a switch, S, we label them #0 through #7. Actually, # is an abbreviation for the index attribute whose values are integers. Since the attribute is a unique character, we do not have to write #:3 (although we could). The additional parameters give information about the physical attributes of the encoding. These are alternative values and any tape has only one of them. A vertical bar (|) indicates this (as in BNF notation for grammars). Thus, 75|112 in/s says that one can have a tape with a speed of 75 inches per second or one with 112 inches per second, but not a tape which can be switched dynamically to run at either speed.

For many of the components no further information is given. Thus, knowing that M.magnetic␣tape is connected to a control and from there to the Pc tells generally what that K does. It is a "tape controller" which evokes all the actions of the tape, such as read, write, rewind; and therefore these actions do not have to be done by Pc. The fact that there is only one K for many Ms's implies that only one tape can be accessed at a time. Other information could be given, although that just provided is all that is usual in specifying a controller in an overall description of a system.

We have used several different ways of saying the same thing in Figure 3 in order to show the range of descriptive notations. Thus, the 64 Teletypes are shown by describing a single connection through a switch and putting the number of links in the switch above the connecting line.

Consider, finally, the Pc in Figure 3. We have given a few parameters: the number of data types, the number of instructions, and the number of interrupts. These few parameters hardly define a processor. Several other important parameters are easily inferred from the Mp. The basic operation time in a processor is a small multiple of the read time of its Mp. Thus it is predictable that Pc stores and reads information in 2 $\times$ 1.5 μs (one for instruction fetch, one for data fetch). Again, where this is not the case (as in the CDC 6600) it is necessary to say so. Similarly, the word size in the Pc is the same as the word size of the Mp—12 data bits. More generally, the Pc must have instructions that take care of evoking all the components of the PMS structure. These instructions of course do not use the switches and controls as distinct entities; rather, they speak directly to the operation of the M's and T's connected via these switches and controls.

Other summary parameters could have been given for the Pc. None would come close to specifying its behavior uniquely, although to those knowledgeable in computers still more can be inferred from the parameters given. For instance, knowing both the data types available in a Pc and the number of instructions, one can come very close to predicting exactly what the instructions are. Nevertheless, the way to describe a Pc in full detail is not to add larger and larger numbers of summary parameters. It is more direct and more revealing to develop a description at the level of instructions, which is the ISP description.

In summary, a descriptive scheme for systems as complex and detailed as digital computers must have the ability to range from extremely complete to highly simplified descriptions. It must permit highly compressed descriptions as well as extensive ones and must permit the selective suppression or amplification of whatever aspects of the computer system are of interest to the user. PMS attempts to fulfill these criteria by providing simple conventions for detailed description with additional conventions that permit abbreviation and abstractions, almost without limit. The result is a notation that may seem somewhat fluid, especially on first contact in such a brief introduction as this. But once assimilated, PMS seems to allow some of the flexibility of natural language within enough notational controls to enhance communication considerably.

ISP LEVEL OF DESCRIPTION

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is determined by a set of bits in Mp, called the program, and a set of interpretation rules, realized in the processor, that specify how particular bit configurations evoke the operations. Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends solely on the particular program in Mp (and also on the initial state of data). This is the level at which the programmer wants the processor described—and which the programming manual provides—since he himself wishes to determine the program. Thus the ISP (Instruction Set Processor) description must provide a scheme for specifying any set of operations and any rules of interpretation.

Actually, the ISP descriptive scheme need only be general enough to cover some broad range of possibilities adequate for past and current generations of machines along with their likely descendants. As with the PMS level, there are certain restrictions that can be placed on the nature of a computer system, specializing it from the more general concept of a discrete state system. For the PMS level, it processes a medium, called information; it is a system of discrete components linked together by information transfers; and each component is characterized by a small set of operations. Similarly, for the ISP level we can add two more such restrictions, which will in turn provide the shape of its descriptive scheme.

The first specialization is that a program can be conceived as a distinct set of *instructions*. Operationally, this means that some set of bits is read from the program in Mp to a memory within P, called the instruction register, M.instruction/M.i. This set of bits then determines the immediately following sequence of operations. Only a single operation may be determined, as in setting a bit in the internal state of the P; or a substantial number of operations may be determined, as in a "repeat" instruction that evokes a search through Mp. In a typical one or two address machine the number of operations per instruction ranges from 2 to 5. In any event, after this sequence of operations has occurred, the next instruction to be fetched from Mp is determined and obtained. Then, the entire cycle repeats itself.

The above cycle of activity is just the *interpretation cycle*, and the part of the P that performs it is the *interpreter*. The effect of each instruction can be expressed entirely in terms of the information held in memories at the end of the cycle (plus any changes made to the outside world). During execution, operations may have internal states of their own as sequential circuits which are not represented as bits in memories. But by the end of the interpretation cycle, whatever effect is to be carried on to a later time has been staticized in bits in some memory.*

The second additional specialization is on the data operations. A processor's total set of operations can be divided into two parts. One part contains those necessary to operate other components given in the PMS diagram—links, switches, memories, transducers, etc. The operations associated with these components and the extent to which they can be indirectly controlled from P are highly constrained by the basic nature of the

* This description holds true for a *P* with a single active control (the interpreter). Some *P*'s (e.g., the CDC 6600) have several active controls and get involved in "overlapping" several instructions and in reordering operations according to the data and devices available. With these, a more complex statement is required to express the same general restriction we have been stating for simple *P*'s: that the program can be decomposed into a sequence of bit sets (the instructions), each of which has local control over the behavior of the *P* for a limited period of time, with all inter-instruction effects being staticized as bits in *M*'s.

components and their controls. The second part contains those operators associated with a processor's D component. So far we have said nothing at all about them, except to exclude them completely from all PMS components except P. These are the operations that produce bit patterns with new meaning—that do all the "real" processing—or changing of information.* If it weren't for data operators, the system would only just transmit information. As we noted in our original definitions, a P (including a D) is the only component capable of directly changing information. A P can create, modify, and destroy information in a single operation. As we noted earlier, D's are like the primitive components in an analog computer. Later, when we express instruction sets as simple arithmetic expressions, the D's are the primitive operators, e.g., $+$, $-$, $\times$, $/$, $\times 2^n$, $\wedge$, $\vee$, $\oplus$, and concatenation ($\square$), which are evoked by the instruction set interpreter part of a processor.

The specialization is that all the data operations can be characterized as working on various *data-types*. For example, there is a data-type called the signed-integer, and there are data operations that add two signed-integers, subtract them, multiply them, take their absolute value, test for which of two is the greater, etc. A data-type is a compound of two things: the referent of the bit pattern (e.g., that this set of bits refers to an integer in a certain range); and the representation in the bit pattern (e.g., that bit 31 is the sign, and bits 30 to 0 are the coefficients of successive powers of 2 in the binary representation of the integer). Thus, a processor may have several data-types for representing numbers: unsigned-integers, signed-integers, single-precision-floating-point, double-precision-floating-point, etc. Each of these requires distinct operations to process it. On occasion, operations for several data-types may all be encoded into a single instruction from the programmer's viewpoint, as when there is an add instruction with a data-type sub-field that selects whether the data is fixed or floating point. The operations are still separate, no matter how packaged, and so their data-types remain distinct.

With these two additional specializations—instructions and data-types—we can define an ISP description of a processor. A processor is completely described at the ISP level by giving its *instruction-set* and its *interpreter* in terms of its *operations*, *data-types* and *memories*.

Let us first give the instruction-set. The effect of each instruction is described by an *instruction-expression*, which has the form:

condition $\rightarrow$ action-sequence.

The *condition* describes when the instruction will be evoked, and the *action-sequence* describes what transformations of data take place between what memories. The right arrow ($\rightarrow$) is the control action (of a K) of evoking an *operation*.

Since all operations in a computer system result in modifications of bits in memories, each action in a sequence has the form:

memory-expression $\leftarrow$ data-expression

The left arrow ($\leftarrow$) is the transmit operation of a link, and corresponds to the ALGOL assign operation. The left side must describe the memory location that is affected; the right side must describe the information pattern that is to be placed in that memory location. The details of data-expressions and memory expressions are patterned on standard mathematical notation, and are communicated most easily by examples. The same is true of the condition, which is a standard expression involving boolean values and relations among memory contents.

There are two important features of the action-sequence. The first is that each action in the sequence may itself be conditional; i.e., of the form, "condition $\rightarrow$ action-sequence." The second is that some actions are sequentially dependent on each other, because the result of one is used as an input to the other; on other occasions a set of actions are independent, and can occur in parallel. The normal situation is the parallel one. For example, if A and B are two registers, then

$$(A \leftarrow B;\ B \leftarrow A);$$

exchanges the contents of A and B. When sequence is required, the term 'next' is used; thus,

$$(A \leftarrow B;\ \text{next } B \leftarrow A);$$

transfers the contents of B to A and then transfers it back to B, leaving both A and B holding the original contents of B (equivalent to $A \leftarrow B$).

An ISP example using the DEC PDP-8 Pc

The memories, operations, instructions, and data-types all need to be declared for a processor. Again

* In principle, this view that only *D* components do "real" processing is false. It can be shown that a universal Turing Machine can be built from *M*, *S*, *L*, and *K* components. The key operation is the write operation into *M*, which suffices to construct arbitrary bit patterns under suitably controlled switches. Hence, arbitrary data operations can be built up. The stated view is correct in practice in that the data operations provided in a *P* are highly efficient for their bit transformations. Only the foolish add integers in a modern computer by table look up.

these are most easily explained by example, although full definitions exist (Bell and Newell, 1970). Consequently, let us examine the ISP description of the Pc of the PDP-8, given in Figure 4.

Processor state

We first need to specify the memories of the Pc in detail, providing names for the various bits. Thus,

AC⟨0:11⟩ *the accumulator*

is a memory called AC, with 12 bits, labeled from 0 to 11 from the left. Comments are given in italics*—in this case that AC is called the accumulator (by the designers of the PDP-8). Alternatively, we could have used the alias or abbreviation convention:

AC⟨0:11⟩/Accumulator⟨0:11⟩.

* There are a few features of the notation, such as the use of italics, which are not easily carried over into current computer character sets. Thus, the ISP of Figure 4 is a publication language.

DEC PDP-8 ISP Description

```
Pc State
   AC<0:11>                                          Accumulator
   L                                                 Link bit/AC extension for overflow and carry
   PC<0:11>                                          Program Counter
   Run                                               1 when Pc is interpreting instructions or "running"
   Interrupt state                                   1 when Pc can be interrupted; under programmed control
   IO pulse 1; IO pulse 2; IO pulse 4                IO pulses to IO devices

Mp State
Extended memory is not included.
   M[0:7777₈]<0:11>
      Page 0[0:177₈]<0:11> := M[0:177₈]<0:11>        special array of directly addressed memory registers
      Auto index[0:7]<0:11> := Page 0[10₈:17₈]<0:11> special array when addressed indirectly, is incremented by 1

Pc Console State
Keys for start, stop, continue, examine (load from memory), and deposit (store in memory) are not included.
   Data switches<0:11>                               data entered via console

Instruction Format
   Instruction/i<0:11>
      op<0:2>              := i<0:2>                 op code
      indirect bit/ib      := i<3>                   0, direct; 1 indirect memory reference
      page 0 bit/p         := i<4>                   0 selects page 0; 1 selects this page
      page address<0:6>    := i<5:11>
      this page<0:4>       := PC'<0:4>
      PC'<0:11>            := (PC<0:11> -1)
      IO select<0:5>       := i<3:8>                 selects a T or Ms device
      io p1 bit            := i<11>                  these 3 bits control the selective generation of -3 volts,
      io p2 bit            := i<10>                   0.4 μs pulses to I/O devices
      io p4 bit            := i<9>
      sma                  := i<5>                   μ bit for skip on minus AC, operate 2 group
      sza                  := i<6>                   μ bit for skip on zero AC
      snl                  := i<7>                   μ bit for skip on non zero Link

Effective Address Calculation Process
   z<0:11> := (                                      effective
     ¬ib → z";
     ib ∧ (10₈ ≤ z" ≤ 17₈) → (M[z"] ← M[z"] + 1; next);   auto indexing
     ib → M[z"])
   z'<0:11> := (¬ ib → z"; ib → M[z"])
   z"<0:11> := (page 0 bit → this page□page address;    direct address
                ¬page 0 bit → 0□page address)

μ microcoded instruction or instruction bit(s) within an instruction
```

```
Instruction Interpretation Process
   Run ∧ ¬ (Interrupt_request ∧ Interrupt_state) → (          no interrupt interpreter
      instruction ← M[PC]; PC ← PC + 1; next                  fetch
      instruction_execution);                                 execute
   Run ∧ Interrupt_request ∧ Interrupt_state → (              interrupt interpreter
      M[0] ← PC; Interrupt_state ← 0; PC ← 1)

Instruction Set and Instruction Execution Process
   Instruction_execution := (
      and (:= op = 0) → (AC ← AC ∧ M[z]);                     logical and
      tad (:= op = 1) → (L□AC ← L□AC + M[z]);                 two's complement add
      isz (:= op = 2) → (M[z'] ← M[z] + 1; next               index and skip if zero
                         (M[z'] = 0) → (PC ← PC + 1));
      dca (:= op = 3) → (M[z] ← AC; AC ← 0);                  deposit and clear AC
      jms (:= op = 4) → (M[z] ← PC; next PC ← z + 1);         jump to subroutine
      jmp (:= op = 5) → (PC ← z);                             jump
      iot (:= op = 6) → (                                     μ in out transfer, microprogrammed to generate up to 3 pulses
         io_p1_bit → IO_pulse_1 ← 1; next                       to an io device addressed by IO_select
         io_p2_bit → IO_pulse_2 ← 1; next
         io_p4_bit → IO_pulse_4 ← 1);
      opr (:= op = 7) → Operate_execution                     the operate instruction is defined below
                        )                                     end Instruction execution

Operate Instruction Set
The microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetic are defined as a separate
instruction set.
   Operate_execution := (
      cla (:= i<4> = 1) → (AC ← 0);                           clear AC. Common to all operate instructions.
      opr_1 (:= i<3> = 0) → (                                 operate group 1
         cll (:= i<5> = 1) → (L ← 0); next                    μ clear link
         cma (:= i<6> = 1) → (AC ← ¬ AC);                     μ complement AC
         cml (:= i<7> = 1) → (L ← ¬ L); next                  μ complement L
         iac (:= i<11> = 1) → (L□AC ← L□AC + 1); next         μ increment AC
         ral (:= i<8:10> = 2) → (L□AC ← L□AC x 2 {rotate});   μ rotate left
         rtl (:= i<8:10> = 3) → (L□AC ← L□AC x 2^2 {rotate}); μ rotate twice left
         rar (:= i<8:10> = 4) → (L□AC ← L□AC / 2 {rotate});   μ rotate right
         rtr (:= i<8:10> = 5) → (L□AC ← L□AC / 2^2 {rotate})); μ rotate twice right
      opr_2 (:= i<3,11> = 10) → (                             operate group 2
         skip condition ⊕ (i<8> = 1) → (PC ← PC + 1); next    μ AC,L skip test
            skip condition := ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
         osr (:= i<9> = 1) → (AC ← AC ∨ Data switches);       μ "or" switches
         hlt (:= i<10> = 1) → (Run ← 0));                     μ halt or stop
      EAE (:= i<3,11> = 11) → EAF_instruction_execution)      optional EAE description
```

Figure 4—DEC PDP-8 ISP Description

AC corresponds to an actual register in the Pc. However, the ISP does not imply any particular implementation, and names may be assigned to various sets of bits purely for descriptive convenience. The colon is used to denote a range or list of values. Alternatively, we could have listed each bit, separating the bit names by commas, as:

$$AC\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\rangle.$$

Having defined a second memory, L (which has only a single bit), one could define a combined register, LAC, in terms of L, and AC, as:

$$LAC\langle L, 0{:}11\rangle := L\square AC.$$

The colon-equal (:=) is used for definition, and the middle square box (□) denotes concatenation. Note that the bit named L of register LAC corresponds to the 1 bit L register.

Memory state

In dealing with addressed memory, either Mp or various forms of working memory within the processor, we need to indicate multidimensional arrays. Thus,

$$Mp[0:7777_8]\langle 0:11\rangle$$

gives the primary memory as consisting of 7777_8 (i.e., base 8) words of 12 bits each, being addressed as indicated. Such an address does not necessarily reflect the switching structure through which the address occurs, though it often will. (Needless to say, it reflects only addressing space, and not how much actual M is available in a PMS structure.) In general, only memory within the processor will occur as operands of the processor's operators. The one exception is primary memory (Mp), which is defined as a memory external to a P, but directly accessible from it.

In writing memories it is natural to use base 10 for all numbers and to consider the basic i-unit of the memory to be a bit. This is always assumed unless otherwise indicated. Since we used base 8 numbers above for specifying the addressing range, we indicated the change of number base by a subscript, in standard fashion. If a unit of information other than the bit were to be used, we would subscript the angle brackets. Thus,

$$Mp[0:7777_8]\langle 0:1\rangle_{64}$$

reflects the same memory. The choice carries with it, of course, some presumption of organization in terms of base 64 characters—but this would show up in the specification of the operators (and is not true, in fact of the PDP-8). We can also have multi-dimensional memories (i.e., arrays), though no examples are used in Figure 4. These just add the extra dimensions with an extra pair of brackets. For example, a more precise description would have used:

$$Mp[0:7][0:31][0:127]\langle 0:11\rangle$$

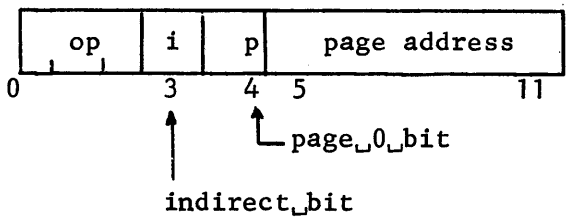
to mean 8 memory fields, each field with 32 pages, each page with 128 words and each word with 12 bits.

Instruction format

It is possible to have several names for the same set of bits; e.g., having defined instruction $\langle 0:11\rangle$ we define the format of the instruction as follows:

op⟨0:2⟩ := instruction ⟨0:2⟩
indirect␣bit := instruction⟨3⟩
page␣0␣bit := instruction⟨4⟩
page␣address⟨0:6⟩ := instruction⟨5:11⟩

The colon-equal (:=) is used to assign names to various parts of the instruction. In effect, this is a definition equivalent to the conventional diagram for the instruction:



Notice that in page␣address the names of all the bits have been shifted, e.g., page␣address⟨4⟩ := instruction⟨9⟩.

In general, a *name* can be any combination of upper and lower case letters and numerals; not including names which would be considered numbers (integers, mixed numbers, fractions, etc.). A compound name can be sequences of names separated by spaces ( ) or a hyphen. In order to make certain compound names more recognizable, a space symbol (␣) may optionally be used to signify the non-printing character.

The instruction set

With all the registers defined, the instructions can be given. These are shown on the second page of Figure 4. The second page is a single expression, named Instruction␣execution, which consists of a list of instructions. These are listed vertically down the page for ease of reading. Each instruction consists of a condition and an action sequence, separated by the condition-arrow (→). In this case the condition is an expression of the form (op = octal-digit). Since op is instruction⟨0:2⟩, this expresses the condition that the operation code of the instruction has a particular value. Each condition has been given a name in passing; e.g., 'and' is the name of (op = 0). This provides the correspondence between the operation code and the mnemonic name of the operation code. If this correspondence had been established elsewhere, or if we didn't care what numerical operation code the "and" instruction is, we could have written:

$$\text{and} \rightarrow (AC \leftarrow AC \wedge M[z])$$

We would not have known what condition the name 'and' stood for, but could have surmised (with little difficulty) that it was simply an equality test on the operation code. Or we could define it elsewhere as:

$$\text{and} := (op = 0)$$

Most generally the form of an instruction is written as:

two's␣complement␣add/tad(:= op = 1) →

(L□AC ← L□AC + M[z])

Here, we simultaneously define the action of the tad instruction, its name, an abbreviation for the name, and the conditions for tad's execution. The first parentheses are, in effect, a remark to allow an in-line definition.

The instructions in the list constitute the total instruction repertoire of the Pc. Since all the conditions are disjoint, one and only one condition will be satisfied when a given instruction is interpreted, hence one and only one action sequence will occur. Actually, all operation codes might not be present, so there would be some illegal op codes that would evoke no action sequence. The act of selection is usually called operation decoding. Here, ISP implies no particular mechanism by which this is carried out.

It might be wondered why the conventions are not more stylized—e.g., some sort of table with mnemonic names in one column, bits of the operation code in another, etc. Though standard processors would fit such a stylized scheme, many others would not—e.g., microprogram processors. By making the ISP description a general expression for evoking action-sequences we obtain the generality needed to cover all variations. (Indeed, you will notice that the PDP-8 ISP is a single expression, and that it incorporates two microprogrammed instructions with no difficulty.)

For the action-sequence standard mathematical infix notation is used. Thus we write

$$AC \leftarrow AC \wedge M[z]$$

This indicates that the word in Mp at address z (determined by the expression on page 1 of Figure 4) is anded with the accumulator and the result left in the accumulator. Each processor will have a basic set of operations that work on data-types of the machine. Here the data-type is simply the 12 bit word viewed as an array of bits.

Operators need not necessarily involve memories actually within the Pc (the processor state). Thus,

$$Mp[z] \leftarrow Mp[z] + 1$$

expresses a change in a word in Mp directly. That this must be mechanized in the PDP-8 by means of some temporary register in Pc is irrelevant to the ISP description.

We also use functional notation, e.g.,

$$AC \leftarrow abs(AC)$$

replaces the contents of the AC with its absolute value.

Effective address calculation

In the examples just given we used z as the address in Mp. This is the effective address (simplified) and is defined as a conditional expression (in the manner of ALGOL or LISP):

$$z\langle 0{:}11\rangle := (\neg\,\text{indirect-bit} \rightarrow z';\ \text{indirect-bit} \rightarrow Mp[z'])$$

The right arrow (→) is the same conditional sign used in the main instruction, similar to the "if . . . then . . ." of ALGOL. The parentheses are used to indicate grouping in the usual fashion. However, we arrange expressions on the page to make reading easier.

As the expression for z shows, we permit conditional within conditionals, and also the nesting of definitions (z is defined in terms of a variable z'). Again, we should emphasize that the structure of such definitions may reflect directly the underlying hardware organization, but it need not. When describing existing processors the ISP description often does or can be forced to reflect the hardware. But if one were designing a processor, then ISP expressions would be put down as design objectives to be implemented in a register transfer structure, which might differ considerably.

Special note should be taken of the opr instruction (op = 6) in Figure 4, since it provides a microprogramming feature. There are two separate options depending on instruction⟨3⟩ being 0 or 1. But common to both of these is the operation of clearing the AC (or not), associated with instruction ⟨4⟩. Then, within one option (instruction⟨3⟩ = 0) there are a series of independently executable actions (following the clearing of L); within the other (instruction⟨3⟩ = 1), there are three independently settable control actions. The nested conditionals and the use of 'next' to force sequential behavior make it easy to see exactly what is going on (in fact a good deal easier than describing it in natural language, as we have been doing).

The instruction interpreter

From the hardware point of view, an interpreter consists of the mechanisms for fetching a new instruction, for decoding that instruction and executing the operations so designated, and for determining the next instruction. A substantial amount of this total job has already been taken care of in the part of the ISP that we have just explained. Each instruction carries with it a condition that amounts to one fragment of the decoding operation. Likewise, any further decoding of the instruction that might be done in common by the interpreter (rather than by the indi-

vidual operation circuits) is implied in the expressions for each instruction, and by the expression for the effective address. The interpreter then fetches the next instruction and executes it.

In a standard machine, there is a basic principle that defines operationally what is meant by the "next instruction." Normally the current instruction address is incremented by one, but other principles are used (e.g., on a processor with a cyclic Mp). In addition, several specific operations exist in the repertoire that can affect what program is in control. The basic principle acts like a default condition—if nothing specific happens to determine program control, the normal "next" instruction is taken. Thus, in the PDP-8 we get an interpretation process that is the classic fetch-execute cycle:

Run → (instruction ← Mp[PC]; PC ← PC + 1;
next Instruction␣execution)

The sequence is evoked so long as Run is true (i.e., its bit value is 1). The processor will simply cycle through the sequence, fetching, then executing the instruction. In the PDP-8 there exists a halt operation that sets Run to be 0, and the console keys can, of course, stop the computer. It should be noted that this ISP description does not include console behavior, although it could.

The ISP description does not determine the way the processor is to be organized to achieve this sequencing, or to take advantage of the fact that many instructions lead to similar sequences. All it does is specify what operations must be carried out for a program in Mp. The ISP description does specify the actual format of the instruction and how it enters into the total operation, although sometimes indirectly. For example, in the case of the *and* operation (op = 0), the definition of AC shows that the AC does not depend on the instruction and the definition of z shows that z does depend on other fields of the instruction (indirect␣bit, page␣0␣bit, page␣address). Likewise, the form of the ISP expression shows that AC and PC both enter into the instruction implicitly. That is, in the ISP description all dependence on memory is explicit.*

Data-types and data operations

Each data-type has a set of operations that are proper to it. Add, subtract, multiply and divide are all proper to any numerical data-type, as well as absolute value and negation. Not all of these need exist in a computer just because it has the data-type, since there are several alternative bases, as well as some levels of completeness. For instance, notice that the PDP-8 first of all does not have multiply and divide (unless one has its special option), thus having a relatively minimal level of arithmetic operations; and second, it does not have a subtract operation, using a two's complement add, which permits negation (−AC) to be accomplished by complementation (∧AC) followed by add 1.

The PDP-8, unlike larger C's, does not have several data representations for what is, externally considered, the same entity. An operator that does a floating add and one that does an integer add are not the same. However, we denote both by the same symbol (in this case, +), indicating the difference parenthetically after the expression. Alternatively, the specification of the data-type can be attached to the data. Thus, in the IBM 7094 we would see the following add instructions:

Add/ADD ← (AC ← AC + M[e]);
Add and Carry Logical/ACL → (AC ← AC + M[e]{sl}).
Floating add/FAD → (AC ← AC + M[e]{sf});
Un-normalized floating add/UFA →
(AC ← AC + M[e]{suf});
Double precision floating add/DFAD →
(ACMQ ← ACMQ + M[e]□M[e + 1]{df});
Double precision un-normalized floating add/DUFA →
(ACMQ ← ACMQ + M[e]□M[e + 1]{duf})

The braces { } differentiate which operation is being performed. Thus above, the data-type* is enclosed in the braces and refers to all the memory elements (operands) of the expression. Alternatively, we also use braces as a modifier to signify the encoding of the i-unit. For example, a fixed point to floating point data conversion operation would be given:

AC{floating} ← AC{fixed}.

We also use the braces as a modifier for the operation type. For example, shifting (left or right) can be a

* This is not correct, actually. In physically realizing an ISP description, additional memories may be utilized (they may even be necessary). It can be said that the ISP description has these memories implicitly. However, it is the case that a consistent and complete description of an ISP can be made without use of these additional memories; whereas with, say, a single address machine, it does not seem possible to describe each instruction without some reference to the implicit memories—as we see in the effective address calculation procedures where definitions look much like registers.

* The conventions for naming data-types is a concatenation of precision, a name and a structure. Examples include i/integer; di/double integer; div/double integer vector; single floating/sf; suf/single unnormalized floating; bv/boolean vector; ch.string/character string.

multiplication or division by a base, but it is not always an arithmetic operation. In the PDP-8, for instance, we had

$$L\square AC \leftarrow L\square AC \times 2\{\text{rotate}\};$$

where the end bits L and AC⟨11⟩ are connected when a shift occurs (the operator is also referred to as a circular shift), or equivalently

$$(L\square AC \leftarrow L\square AC \times 2;\ AC\langle 11\rangle \leftarrow L).$$

In general, the nature of the operations used in processors are sufficiently familiar that no definitions are required, and they can all be taken as primitive. It is only necessary to have agreed upon conventions for the different data representations used. In essence, a data-type is made up recursively of a concatenation of subparts, which themselves are data types. This concatenation may be an iteration of a data-type to form an array.

If required, an operation can be defined in terms of other (presumably more primitive) operations. It is necessary, of course, first to define the data format explicitly (including perhaps some additional memory). Variables for the operands are permitted in the natural way. For example, binary single precision floating point multiplication on a 36 bit machine could be defined in terms of the data fields as follows:

```
sf mantissa/mantissa      := ⟨0:27⟩
sf sign/sign              := ⟨0⟩
sf exponent/exponent      := ⟨28:35)
sf exponent␣sign          := ⟨28⟩
x1 ← x2 × x3{sf}          := (x1 mantissa := x2 mantissa × x3 mantissa;
                              x1 exponent := x2 exponent + x3 exponent; next
                              x1 := normalize(x1){sf})
```

where normalize is:

```
x1 ← normalize(x2) {sf}  := (
     (x1 mantissa = 0)  → (x1 exponent := 0)
     (x2 mantissa ≠ 0) ∧ (x2⟨0⟩ = x2⟨1⟩) → (
                  x1 mantissa := x2 mantissa × 2;
                  x1 exponent := x2 exponent-1; next
                  x1 := normalize (x2){sf}))
```

Three additional aspects need to be noted with respect to data-types; two substantive, one notational. First, not everything one does with an item of data makes use of all the properties of its data-type. For example, numbers have to be moved from place to place. This operation is not a numerical operation, and does not depend on the item being a number. Second, one can often embed one kind of operation in another, so as to coalesce data-types. An example is encoding the Mp addresses into the same integer data-type as are used for regular arithmetic. Then there need be no separate data-type for addresses.*

The notational aspect is our use in ISP of an mnemonic abbreviation scheme for data-types. We have already used sf for single-precision-floating-point. More generally, an abbreviation is made up of a letter showing the length, a letter showing the type, and a letter showing the structure. The simple naming convention does not take into account all we know about a data-type. The information carrier for the data is only partially included in the length characteristic. Thus the carrier should also include the data base and the sign convention for representing negative numbers, (e.g., sign-magnitude).

PMS structure of the CDC 6600 series

A simplified PMS structure of the C('6400|'6600) is given in Figure 5. Here we see the C(io; #1:10) each of which can access the primary memory (Mp) of the central computer (Cc). Figure 5 shows why one considers the 6600 to be a network. Each Cio (actually a general purpose, 12 bit C) can easily serve the specialized Pio function for Cc. The Mp of Cc is an Ms for a Cio because the Cio cannot execute programs from this memory. By having a powerful Cio more complex input-output tasks can be handled without Cc intervention. These tasks can include data-type conversion, error recovery, etc. The K's which are connected to a Cio can also be less complex.

A detailed PMS diagram for the C('6400, '6416, '6500, *and* '6600) is given in Figure 6. The interesting structural aspects can be seen from this diagram. The four configurations, 6400 ~ 6600, are included just by considering the pertinent parts of the structure. That

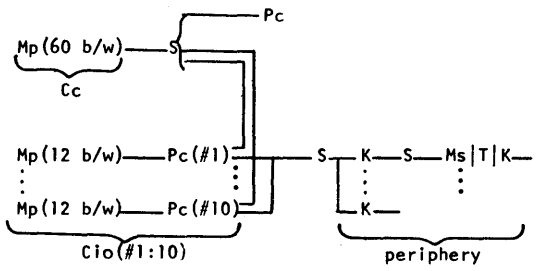
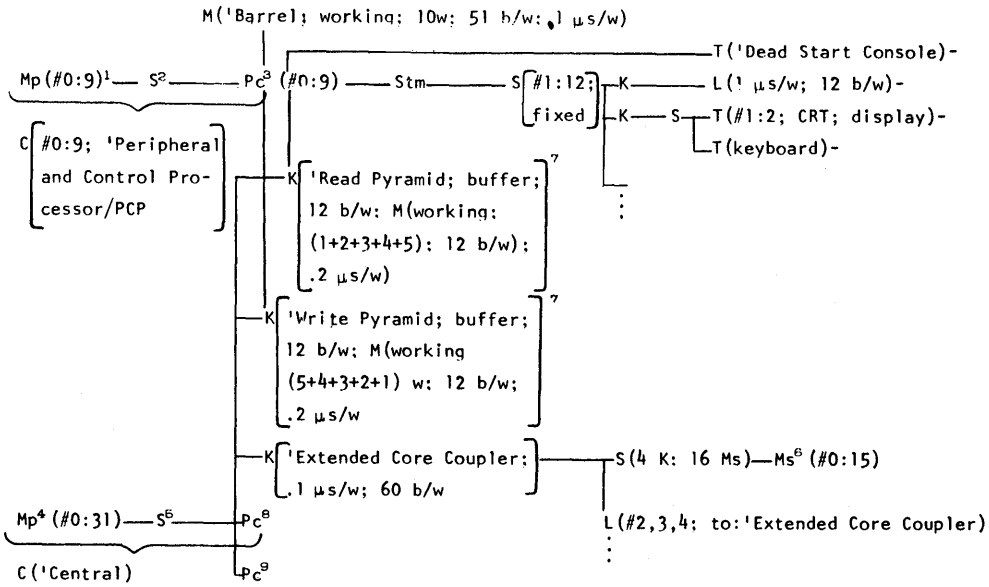


Figure 5—CDC 6600 PMS diagram (simplified)

* However logical such a course may seem, it is not always done this way. For example, the IBM 7090 (and other members of that family) have a 15 bit address data type and a 36 bit integer data type, with separate operations for each.



[1] Mp(core; 1.0 μs/w; 4096 w; 12 b/w)

[2] S(time multiplex; .2 μs/w; 12 b/w)

[3] Pc('Peripheral and Control Processor; #0:9; time multiplex:.1 μs/w; 1 address/instruction; 12 b/w; Mps('Program Counter, Accumulator) 1,2 w/instruction)

[4] Mp(core; 1.0 μs/w; 4096 w; (5 x 12) b/w)

[5] S(time multiplex; 0.1 μs/w; 60 b/w)

[6] Ms('Extended Core Storage/ECS; 3.2 μs/w; (125952 / 8) w; (8 x (60, 1 parity)) b/w)

[7] See Chapter 39 for operation.

[8] Only present in CDC 6500

[9] No C('Central) in CDC 6416; CDC 6500 and CDC 6400 do not have K('Scoreboard), separate D's, and M('Instruction Stack).

Pc('6600; 15, 30 b/instruction; technology:transistor; ~ 1964; data: si,bv,w,sf,df) :=

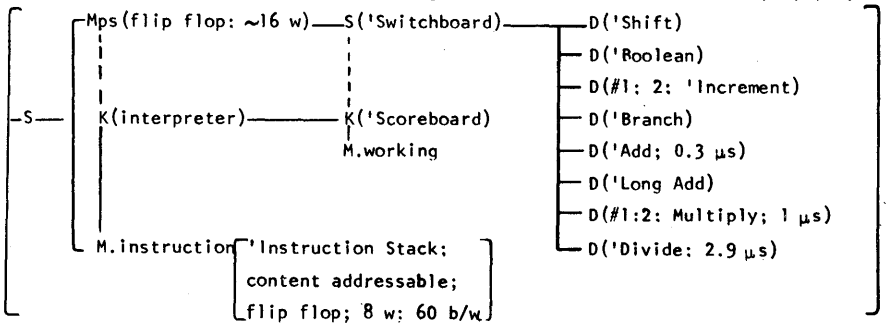


Figure 6—CDC 6600 PMS Diagram

is, a 6416 has no large Pc; a 6400 has a single straightforward Pc; a 6500 has two Pc's; and the 6600 has a single powerful Pc. The 6600 Pc has 10 D's, so that several parts of a single instruction stream can be interpreted in parallel. A 6600 Pc also has considerable M.buffer to hold instructions so that Pc need not wait for Mp fetches.

The implementation of the 10 Cio's can be seen from the PMS diagram (Figure 6). Here, only 1 physical processor is used on a time shared basis. Each 0.1 μs a new logical P is processed by the physical P. The 10 Mp's are phased so that a new access occurs each 0.1 μs. The 10 Mp's are always busy. Thus, the information rate is (10 × 12) b/μs or 120 megabit/s.

This structure for shifting a new Pc state into position each 0.1 μs has been likened by CDC to a barrel.

The T's, K's and M's are not given in the figures, although it should be mentioned that the following units are rather unique: a K for the management of 64 telegraph lines to be connected to a Cio; and Ms(disk) with four simultaneous access ports, each at 1.68 megachar/s data transfer rate; and a capacity of 168 megachar; a Ms(magnetic tape) with a K(#1:4) and S to allow simultaneous transfers to 4 Ms; the

```
CDC 6400, 6500, 6600 Central Processor ISP Description

Pc State
   P<17:0>                                  Program counter
   X[0:7]<59:0>                             Main arithmetic registers. X[1:5], are implicitly loaded from
   A[0:7]<17:0>                              Mp when A[1:5] are loaded.  X[6:7] are implicitly stored in
                                             Mp when A[6:7] are loaded.
   B[0]<17:0>  := 0                         B registers are general arithmetic registers, and can be used
   B[1:7]<17:0>                              as index registers.
   Run                                      1 if interpreting instructions, not under program control.
   EM<17:0>                                 Exit mode bits
      Address_out_of_range_mode  := EM<12>
      Operand_out_of_range_mode  := EM<13>
      Indefinite_operand_mode    := EM<14>
 The above description is incomplete in that the above 3 mode's alarm allow conditions to trap Pc at Mp[RA].  Trapping occurs if
 an alarm condition occurs "and" the mode is a one.

Mp State
   Mp[0:777777(8)]<59:0>                    main core memory of 2^18 w, (256 kw)
   Ms[0:2015232]<59:0>                      ECS/Extended Core Storage Program can only transfer data between
                                             Mp and Ms.  Program cannot be executed in Ms.
   RA<17:0>                                 reference (or relocation) address register to map a logical Mp'
                                             into physical Mp
   FL<17:0>                                 field length - the bounds register which limits a program's
                                             access to a range of Mp'
   RAECS<59:36>                             reference or relocation register for Ms(Extended Core Storage)
   FLECS<59:36>                             field length for ECS
   Address_out_of_range                     a bit denoting a state when memory mapping is invalid

Memory Mapping Process
 This process maps or relocates a logical program, at location Mp', and Ms', into physical Mp and Ms.
   Mp'[X]  := ((X < FL) →Mp[X + RA]);       logical Mp'
              (X ≥ FL) →(Run ←0; Address_out_of_range ←1))
   Ms'[X]  := ((X < FLECS) →Ms[X]+ RAECS]); logical Ms'
              (X ≥ FLECS) →(Run ←0; Address_out_of_range ←1))

Exchange jump storage allocation map at location, n within Mp:
 The following Mp" array is reserved when Pc state is stored, and switched to another job.  The exchange jump instruction in
 a Peripheral and Control Processor enacts the operation:  (Mp" ← Mp; Mp ← Mp").
   Mp"[n]<53:0>     := P□A[0]□000000(8)
   Mp"[n+1]<53:0>   := RA□A[1]□B[1]
   Mp"[n+2]<53:0>   := FL□A[2]□B[2]
   Mp"[n+3]<53:0>   := EM□A[3]□B[3]
   Mp"[n+4]         := RAECS□A[4]□B[4]
   Mp"[n+5]         := FLECS□A[5]□B[5]
   Mp"[n+6]<35:0>   := A[6]□B[6]
   Mp"[n+7]<35:0>   := A[7]□B[7]
   Mp"[n+10(8):n+17(8)]:= X[0:7]
```

Figure 7—CDC 6400, 6500, 6600 Central Processor ISP Description

T(direct; display) for monitoring the system's operation; K's to other C's and Ms's; and conventional T(card reader, punch, line-printer, etc.).

ISP OF THE CDC 6600

The ISP description of the Pc is given in Figure 7. The Pc has a straightforward scientific calculation

```
Instruction Format
   instruction<29:0>                                  although 30 bits, most instructions are 15 bits; see
                                                       Instruction Interpretation Process
      fm<5:0>          := instruction<29:24>          operation code or function
      fmi<8:0>         := fm□i                        extended op code
      i<2:0>           := instruction<23:21>          specifies a register or an extension to op code
      j<2:0>           := instruction<20:18>          specifies a register
      k<2:0>           := instruction<17:15>          specifies a register
      jk<5:0>          := j□k                         a shift constant (6 bits)
      K<17:0>          := instruction<17:0>           an 18 bit address size constant
      long_instruction := ((fm < 10_8) ∨              30 bit instruction
                           (50 ≤ fm ≤ 53) ∨
                           (60 ≤ fm ≤ 63) ∨
                           (70 ≤ fm ≤ 73))
   short_instruction   := ¬ long instruction          15 bit instruction

Instruction Interpretation Process
A 15 bit (short) or 30 bit (long) instruction is fetched from Mp'[P]<p x 15 + 15 - 1:p x 15> where p = 3, 2, 1, or 0. A 30
bit instruction cannot be stored across word boundaries (or in 2, Mp' locations).
   p<1>_4                                             a pointer to 15 bit quarter word which has instruction
   Run → (instruction<29:15> ← Mp'[P]<(p x 15 + 14):(p x 15)>; next      fetch
      p ← p - 1; next
      (p = 0) ∧ long_instruction → Run ← 0;
      (p ≠ 0) ∧ long_instruction → (
         instruction<14:0> ← Mp'[P]<(p x 15 + 14):(p x 15)>;
            p ← p - 1); next
      Instruction_execution; next                                    execute
      (p = 0) → (p ← 3; P ← P + 1))

Instruction Set and Instruction Execution Process
Operand fetches or stores between Mp' and X[i] occur by loading or storing registers A[i]. If (0 < i < 6) a fetch from
Mp'[A[i]] occurs. If (i ≥ 6) a store is made to Mp'[A[i]]. The description does not describe Address_out_of_range case,
which is treated like a null operation.
   Instruction_execution := (
   Set A[i]/CA
      "SAi Aj + K" (fm = 50) → (A[i] ← A[j] + K; next Fetch_Store);
      "SAi Bj + K" (fm = 51) → (A[i] ← B[j] + K; next Fetch_Store);
      "SAi Xj + K" (fm = 52) → (A[i] ← X[j]<17:0> + K; next Fetch_Store);
      "SAi Xj + Bk" (fm = 53) → (A[i] ← X[j]<17:0> + B[k]; next Fetch_Store);
      "SAi Aj + Bk" (fm = 54) → (A[i] ← A[j] + B[k]; next Fetch_Store);
      "SAi Aj - Bk" (fm = 55) → (A[i] ← A[j] - B[k]; next Fetch_Store);
      "SAi Bj + Bk" (fm = 56) → (A[i] ← B[j] + B[k]; next Fetch_Store);
      "SAi Bj - Bk" (fm = 57) → (A[i] ← B[j] - B[k]; next Fetch_Store);
      Fetch_Store := (
         (0 < i < 6) → (X[i] ← Mp'[A[i]]);                  process to get operand in X or store operand from X when A
         (i ≥ 6) → (Mp'[A[i] ← X[i]))                        is written
   Operations on B and X
   Set B[i]/SBi
      "SBi Aj + K" (fm = 60) → (B[i] ← A[j] + K);
```

Figure 7 (Continued)

oriented ISP with 48 bit mantissa single precision floating point (also double precision floating point operations is provided). The Pc state has three sets of general registers. This structure assumes that a program consists of several read accesses to a large array(s), a large number of operations on these accessed ele-

```
    "SBi Bj + K" (fm = 61) → (B[i] ← B[j] + K);
    "SBi Xj + K" (fm = 62) → (B[i] ← X[j]<17:0> + K);
    "SBi Xj + Bk" (fm = 63) → (B[i] ← X[j]<17:0> + B[k]);
    "SBi Aj + Bk" (fm = 64) → (B[i] ← A[j] + B[k]);
    "SBi Aj - Bk" (fm = 65) → (B[i] ← A[j] - B[k]);
    "SBi Bj + Bk" (fm = 66) → (B[i] ← B[j] + B[k]);
    "SBi Bj - Bk" (fm = 67) → (B[i] ← B[j] - B[k]);
Set X[i]/SXi
    "SXi Aj + K" (fm = 70) → (X[i] ← sign_extend(A[j] + K));
    "SXi Bj + K" (fm = 71) → (X[i] ← sign_extend(B[j] + K));
    "SXi Xj + K" (fm = 72) → (X[i] ← sign_extend(X[j] + K));
    "SXi Xj + Bk" (fm = 73) → (X[i] ← sign_extend(X[j] + B[k]));
    "SXi Aj + Bk" (fm = 74) → (X[i] ← sign_extend(A[j] + B[k]));
    "SXi Aj - Bk" (fm = 75) → (X[i] ← sign_extend(A[j] - B[k]));
    "SXi Bj + Bk" (fm = 76) → (X[i] ← sign_extend(B[j] + B[k]));
    "SXi Bj - Bk" (fm = 77) → (X[i] ← sign_extend(B[j] - B[k]));
Miscellaneous program control
    "PS" (:= fm = 0) → (Run ← 0);                                  program stop
    "NO" (:= fm = 46) → ;                                          no operation; pass
Jump unconditional
    "JP Bi + K" (:= fm = 02) → (P ← B[i] + K; p ← 3);              jump
Jump on X[j] conditions
    "ZR Xj K" (:= fmi = 030) → ((X[j] = 0) → (P ← K; p ← 3));      zero
    "NZ Xj K" (:= fmi = 031) → ((X[j] ≠ 0) → (P ← K; p ← 3));      non zero
    "PL Xj K" (:= fmi = 032) → ((X[j] ≥ 0) → (P ← K; p ← 3));      plus or position
    "NG Xj K" (:= fmi = 033) → ((X[j] < 0) → (P ← K; p ← 3));      negative
    "IR Xj K" (:= fmi = 034) → (                                   out of range constant tests
    ¬((X[j]<59:48>= 3777)∨ (X[j]<59:48> 4000)) →P ← K; p ← 3);
    "OR Xj K" (:= fmi = 035) → (
       (X[j]<59:48>=3777) ∨ (X[j]<59:48>=4000)→ (P ← K; p ← 3));
    "DF Xj K" (:= fmi = 036) → (                                   indefinite form constant tests
       (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000)  →(P ← K; p ← 3));
    "ID Xj K" (:= fmi = 037) → (
       (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000)   (P ← K; p ← 3));
Jump on B[i], B[j] comparison
    "EQ Bi Bj K" (:= fm = 04) →((B[i] = B[j]) →(P ← K; p ← 3));    equal
    "NE Bi Bj K" (:= fm = 05) →((B[i] ≠ B[j]) →(P ← K; p ← 3));    not equal
    "GE Bi Bj K" (:= fm = 06) →((B[i] ≥ B[j]) →(P ← K; p ← 3));    greather than or equal
    "LT Bi Bj K" (:= fm = 07) →((B[i] < B[j]) →(P ← K; p ← 3));    less than
Subroutine call
    "RJ K" (:= fmi = 010) →(                                       return jump
       M[K]<59:30> ←04₈□00₈□(P + 1)□000000₈; next
       (P ←K + 1; p ←3));
Reading (REC) and writing (WEC) Mp with Extended Core Storage, subjected to bounds checks, and Ms', Mp' mapping
    "REC Bj + K" (:= fmi = 011) →(                                 read extended core
```

Figure 7 (Continued)

ments, followed by occasional write accesses to store results.

Cc has provisions for multiprogramming in the form of a protection and relocation address. The mapping is given in the ISP description for both Mp, but an Ms('Extended Core Storage/ECS) is not described.

```
    Mp'[A[0]:A[0] + B[j] + K-1] ← Ms'[X[0]:X[0] + B[j] + K-1];
  "WFC Bj + K" (:= fmi = 012) → (                                  write extended core
    Ms'[X[0]:X[0] + B[j] + K-1]   Mp'[A[0]:A[0] + B[j] + K-1]);
Fixed Point Arithmetic and Logical operations using X
  "IXi Xj + Xk" (:= fm = 36)  →(X[i] ← X[j] + X[k]);               integer sum
  "IXi Xj - Xk" (:= fm = 37)  →(X[i] ← X[j] - X[k]);               integer difference
  "CXi Xk" (:= fm = 47)  →(X[i] ← sum_modulo_2(X[k]);              count the number of bits in X[k]
  "BXi Xj" (:= fm = 10₈)  →(X[i] ← X[j]);                          transmit
  "BXi Xj * Xk" (:= fm = 11₈)  →(X[i] ← X[i] ←X[j] ∧ X[k]);        logical product
  "BXi Xj + Xk" (:= fm = 12)  →(X[i] ← X[j] ∨ X[k]);               logical sum
  "BXi Xj - Xk" (:= fm = 13)  →(X[i] ←X[j] ⊕ X[k]);                logical difference
  "BXi - Xk" (:= fm = 14)  →(X[i] ←¬ X[k]);                        transmit complement
  "BXi - Xk * Xj" (:= fm = 15)  →(X[i] ←X[j] ∧ ¬ X[k]);            logical product and complement
  "BXi - Xk + Xj" (:= fm = 16)  →(X[i] ←X[j] ∨ ¬ X[k]);            logical sum and complement
  "BXi - Xk - Xj" (:= fm = 17)  →(X[i] ←X[j] ⊕ ¬ X[k]);            logical difference and complement
  "LXi jk" (:= fm = 20)  →(X[i] ← X[i] x 2^jk {rotate});
  "AXi jk" (:= fm = 21)  →(X[i] ←X[i] / 2^jk);                     arithmetic right shift
  "LXi Bj Xk" (:= fm = 22) →(                                      left shift nominally
    ¬B[j]<17> →X[i] ←X[k] x 2^B[j]<5:0> {rotate};
     B[j]<17> →X[i] ←X[k] / 2^¬ B[j]<10:0>);
  "AXi Bj Xk" (:= fm = 23) →(                                      arithmetic right shift nominally
    ¬B[j]<17> →X[i]←X[k] / 2^B[j]<10:0>;
     B[j]<17> →X[i] ← X[k] x 2^¬ B[j]<5:0> {rotate});
  "MXi jk" (:= fm = 43) →(                                         form mask
    X[i]<59:59-jk+1> ← 2^jk - 1;
    (jk = 0) →X[i] ← 0);
Floating Point Arithmetic using X
 Only the least significant (lo) part of arithmetic is stored in Floating DP operations.
  "FXi Xj + Xk" (:= fm = 30) →(X[i] ←X[j] + X[k] {sf});            floating sum
  "FXi Xj - Xk" (:= fm = 31) →(X[i] ←X[j] - X[k] {sf});            floating difference
  "DXi Xj + Xk" (:= fm = 32) →(X[i] ←X[j] + X[k] {ls.df});         floating dp sum
  "DXi Xj - Xk" (:= fm = 33) →(X[i] ←X[j] - X[k] {ls.df});         floating dp difference
  "RXi Xj + Xk" (:= fm = 34) →(
    X[i] ← round(X[j]) + round(X[k]) {sf});
  "RXi Xj - Xk" (:= fm = 35) →(                                    round floating difference
    X[i] ← round(X[j]) - round(X[k]) {sf});
  "FXi Xj * Xk" (:= fm = 40) →(X[i] ←X[j] x X[k] {sf});            floating product
  "RXi Xj * Xk" (:= fm = 41) →(                                    round floating product
    X[i] ← X[j] x X[k] {sf}; next X[i] ← round(X[i]) {sf});
  "DXi Xj * Xk" (:= fm = 42) →(X[i] ← X[j] x X[k] {ls.df});        floating dp product
  "FXi Xj / Xk" (:= fm = 44) →(X[i] ← X[j] / X[k] {sf});           floating divide
  "RXi Xj / Xk" (:= fm = 45) →(X[i] ← round(X[j] / X[k]) {sf});    round floating divide
  "NXi Bj Xk" (:= fm = 24) →(                                      normalize
    X[i] ← normalize(X[k]) {sf};
    B[j] ← normalize_exponent(X[k]) {sf});
```

Figure 7 (Continued)

```
"ZXI BJ Xk" (:= fm = 25) → (                                    round and normalize
   X[i] ← round(X[k]) {sf}; next
   X[i] ← normalize(X[i]) {sf};
   B[j] ← normalize_exponent(X[i]) {sf});
"UXi BJ Xk' (:= fm = 26) → (B[j] ← X[k]<58:48> {si};            unpack
                            X[i] ← X[k]<59,47:0> {si});
"PXi BJ Xk" (:= fm = 27) → (X[k]<58:48> ← B[j] {si};            pack
                            X[k]<59,47:0> ← X[i] {si});
                         )                                      end Instruction_execution
```

Figure 7 (Continued)

SUMMARY

We have introduced two notations for two aspects of the upper levels of computer systems: the topmost information-flow level, here called the PMS level (there being no other common name); and the interface between the programming level and the register transfer level, called ISP.

We were induced to create these notations as an aid in writing a book describing the architecture of many different computers—which served to make us painfully aware of the (dysfunctional) diversity that now exists in our way of describing systems. It would have been preferable to have notational systems constructed around techniques of analysis or synthesis (i.e., simulation languages). But our immediate need was for adequate descriptive power to present computer systems for a text. Considering the amount of effort it has taken to make these notational systems reasonably polished, it seems to us they should be presented to the computer profession, for criticism and reaction.

The main sources of experience with the notation so far is in the aforementioned book, where we have developed PMS diagrams for 22 systems* and ISP descriptions for 14 systems.** The levels of details in all of these is as adequate as the programming manual, i.e., as complete as the description of the PDP-8 example given here. In addition at least one new machine, the DEC PDP-11 (these proceedings), has made use of the notation at the formulation and design stage.

REFERENCES

1 C G BELL A NEWELL
Computer structures: Readings and examples
In Press McGraw-Hill Company 1970

2 Y CHU
Digital computer design fundamentals
McGraw-Hill Book Company 1962

3 J A DARRINGER
The description, simulation, and automatic implementation of digital computer processors
Thesis for Doctor of Philosophy degree College of Engineering and Science Department of Electrical Engineering Carnegie-Mellon University Pittsburgh Pennsylvania May 1969

4 A D FALKOFF K E IVERSON E H SUSSENGUTH
A formal description of system/360
IBM Systems Journal Vol 3 No 3 pp 198–261 1956

5 T B STEEL JR
A first version of UNCOL
Proceedings WJCC pp 371–377 1961

* ARPA network; Burroughs B5500, B6500; CDC 6600; LGP 30; ComLogNet; DEC LINC-8-338, PDP-11; English Electric Deuce, KDF-9; IBM 1800, 7401, 7094, System/360 (Models 30 ~ 91), ASP network; LRL network; MIT's Whirlwind I; NBS'S Pilot; RW 40, SDS 910 930; UNIVAC 1108.

** The computers and the associated number of description pages (enclosed in parentheses) are CDC 160A (2), 6600 PPU (2), 6600 CPU (4¼); DEC PDP-8 (2, 3 with options), PDP-11 (5), 338 (5), IBM 1800 (3½), 1401 (3½), 7094 CPU (7), 7094 Data Channel (6½); LINC (~3); RW 40 (2½); SDS 92 (~3), 930 (4).

Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair

by FRANCIS P. MATHUR*

Jet Propulsion Laboratory
Pasadena, California

and

ALGIRDAS AVIŽIENIS

University of California
Los Angeles, California

"A random series of inept events
To which reason lends illusive sense, is here,
Or the empiric Lifes instinctive search,
Or a vast ignorant mind's colossal work . . ."
Savitri, B.I.C.2—Sri Aurobindo[1]

INTRODUCTION: FAULT-TOLERANT COMPUTING

The objective to attain fault-tolerant computing has been gaining an increasing amount of attention in the past several years. A digital computer is said to be fault-tolerant when it can carry out its programs correctly in the presence of logic faults, which are defined as any deviations of the logic variables in a computer from the design values. Faults can be either of transient or permanent duration. Their principal causes are: (1) component failures (either permanent or intermitent) in the circuits of the computer, and (2) external interference with the functioning of the computer, such as electric noise or transient variations in power supplies, electromagnetic interference, etc.

Protective redundancy in the computer system provides the means to make its operation fault-tolerant. It consists of additional programs, additional circuits, and additional time of operation that would not be necessary in a perfectly fault-free system. The redundancy is deliberately incorporated into the circuits and/or software of the computer in order to provide either masking of or recovery from the effects of some types of faults which are expected to occur in the computer. Repetition of programs provides time redundancy. Programmed reasonableness checks and diagnostic programs are forms of software redundancy. Finally, monitoring circuits, error-detecting and error-correcting codes, structural redundancy of logic circuits (component quadding, channel triplication with voting, etc.), replication of entire computers, and self-repair by the switching-in of standby spares (replacement systems) are the most common forms of hardware redundancy.

The historical perspective shows that the study and use of hardware redundancy, which began nearly 20 years ago,[2,3] has been steadily increasing in the past decade. A very strong reason for this has been the evolution of integrated circuit technology. The inclusion of redundant circuitry is now economically more feasible. The large cost and size of diagnostic software in today's complex computer systems also motivates the relegation of as much checking as possible to special hardware. This special hardware is required to interact with a supervisory program to provide fault-tolerance and recovery without interaction with the human operator.

The presently existing computer systems with extensive use of hardware redundancy are found in applications with extreme reliability requirements. The

*This work was done in partial fulfillment towards the Ph.D. in the Computer Science Department of the University of California, Los Angeles. A preliminary version of this paper was presented as a working paper at the IEEE Computer Group Workshop on "Reliability and Maintainability of Computing Systems," Lake of the Ozarks, Missouri, October 20–22, 1969.

most interesting illustration is the SATURN V launch vehicle computer which employs triple-modular redundancy (TMR) with voting elements in its central processor and duplication in the main memory.[4] Subsequent studies of fault-tolerance in manually non-accessible computers with life requirements of over 10 years have shown that replacement systems with standby spares of entire computer subsystems offer advantages over complete triplication.[5] These studies have led to the design and construction of an experimental Self-Testing-And-Repairing (STAR) computer.[6] This computer is presently in operation at the Jet Propulsion Laboratory. It is being used as an experimental vehicle to study and refine self-repair techniques which incorporate fault-detection and recovery by repetition of programs and/or by automatic replacement of faulty subsystems.

Many systems with hardware redundancy (including the STAR computer and other replacement-repair systems) share the common problem of a "hard core." This "hard core" consists of logic circuits which must continue to function in real time in order to assure the proper fault detection and recovery of the entire system. The purpose of this paper is to present the results of a general study of the architecture and reliability analysis of a new class of digital systems which are suitable to serve as the "hard core" of fault-tolerant computers. These systems are called *hybrid-redundant* systems and consist of the combination of a multiplexed system with majority voting (providing instant internal fault-masking) and of standby spare units (providing an extended mean life over the purely multiplexed system). The new quantitative results demonstrate that hybrid systems possess advantages over purely multiplexed systems in the relative improvement of reliability and mean life with respect to a nonredundant reference system.

It is also possible that the continuing miniaturization of computers will make hybrid redundancy applicable at the level of an entire computer serving as the non-redundant reference unit. The hybrid-redundant multicomputer system may then serve as the hard core of very large and complex data handling systems, such as those required for spacecraft, automated telephone exchanges, digital communication systems, automated hospital monitoring systems, and time-sharing-utility centers.

TABLE OF SYMBOLS AND NOTATION

λ	Failure rate of a non-redundant active unit, $(\lambda \geq 0)$.
μ	Failure rate of a non-redundant standby-spare unit, $(\mu \leq \lambda)$.
K	Ratio of λ to μ, $(=\lambda/\mu)$, $1 \leq K \leq \infty$.
S	Total number of standby-spare units, $(S \geq 0)$.
N	Total number of active redundant units, $(=2n + 1)$.
n	Degree of active redundancy, $(=(N - 1)/2)$.
C	Total number of units in a system, $(=N + S)$.
T	Mission time, (≥ 0).
t or τ	Dummy variables for time, $(0 \leq t$ or $\tau \leq T)$.
$\binom{A}{B}$	Combinatorial notation for $\frac{A!}{(A - B)!B!}$.
DD	Disagreement detector.
SU	Switching unit.
R–S–D unit	An abbreviation for the unit which incorporates the restoring organ, switching unit, and the disagreement detector.
Simplex system	A non-redundant unit or system.
TMR system	Triple-modularly redundant system, $(N = 3)$.
NMR system	N-tuple-modularly redundant system.
Hybrid (N, S) system	A hybrid redundant system having a total of $N + S$ units of which N units are active and S units are standby-spares.
$H(N, S)$	An abbreviation for Hybrid (N, S).
$H(N, 0)$	A reduced case of $H(N, S)$ which yields an equivalent system to basic NMR under the assumption of fail-proof R–S–D unit and voter elements.
$H(3, 0)$	A reduced case of $H(N, 0)$ which yields an equivalent system to basic TMR.
R ("System Characterization") ["time"]	The format of a compact notation for simplifying the writing of reliability equations. Here "R" the reliability is followed in parentheses by the "system characterization" such as (N, S), (NMR), (TMR) or (Simplex) and is then succeeded in square brackets by the parameter "time." The parameter "time" is usually the mission time T and this term may be omitted if it is unambiguous to do

so. If the "system characterization" refers to a simplex system, then both the "system characterization" term and the "time" term may be omitted.

Thus,

$R(N, S)[T]$ is the reliability of a hybrid redundant system $H(N, S)$ for a mission time of duration T.

THE N-TUPLY MODULAR REDUNDANT SYSTEM

The basic TMR types of systems are first reviewed and are illustrated in Figure 1. A simplex or nonredundant system having reliability R is shown in Figure 1(a). The reliability of the basic triple-modular or TMR system as shown in Figure 1(b) is given (under the worst-case assumption that no compensating failures occur) by the following well known equation:

$$R(\text{TMR}) = R^3 + 3R^2(1 - R) \qquad (1)$$

The generalization of the TMR concepts[7] to an N-tuply modular system utilizing $N = 2n + 1$ units and having a $(n + 1)$ out-of-n-restoring organ is illustrated in Figure 1(c) and is therein designated as the

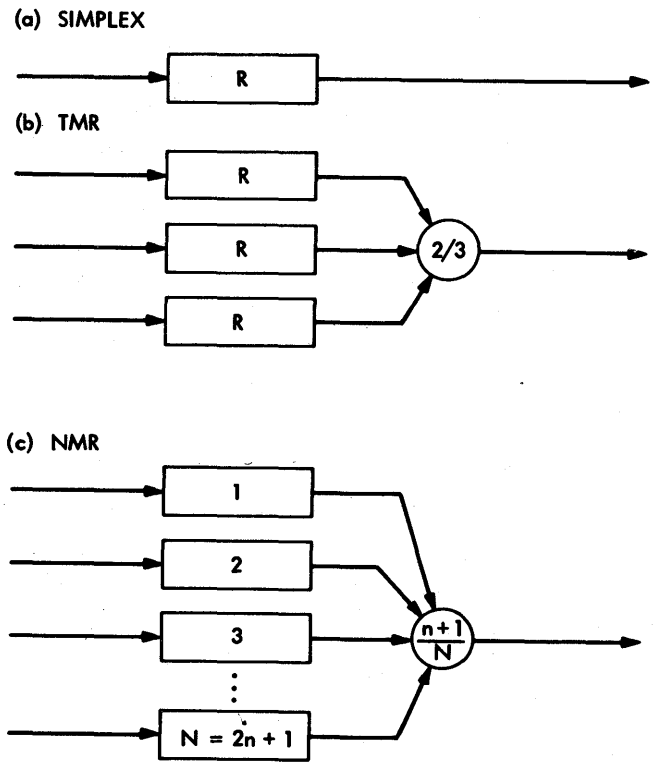


Figure 1—Basic TMR-type systems

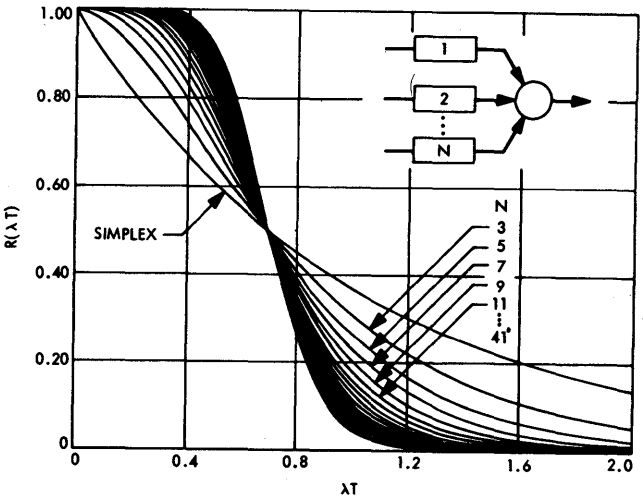


Figure 2—Reliability of NMR-type systems vs normalized time

NMR system; its reliability equation is

$$R(\text{NMR}) = \sum_{i=0}^{n} \binom{N}{i} (1 - R)^i R^{N-i} \qquad (2)$$

where the combinatorial notation $\binom{N}{i} = \dfrac{N!}{(N - i)!\,i!}$

The family of curves illustrating its behavior is shown in Figure 2, with reliability plotted as a function of normalized time λT. The underlying failure law throughout this paper is assumed to be exponential.[8] Thus the simplex reliability R is given by $\exp(-\lambda T)$, where λ is the failure rate of the nonredundant system when it is active. In the ensuing development of the probabilistic model for the Hybrid(N,S) systems, the assumption of statistical independence of failures has been made.

THE HYBRID(N,S) SYSTEM

The Hybrid(N,S) system concept (Figure 3) consists of an NMR core, with an associated bank of S spare units such that when one of the N active units fails, the spare unit replaces it and restores the NMR core to the all-perfect state. The active NMR units have a failure rate designated by λ, while the standby-spare units, which are said to be in a dormant mode,[9] have a failure rate designated by μ $(\mu \leq \lambda)$, with the corresponding reliability $R_s = \exp(-\mu T)$.

The physical realization of such a system is shown in Figure 4, where the disagreement detector (DD) compares the system output from the restoring organ with

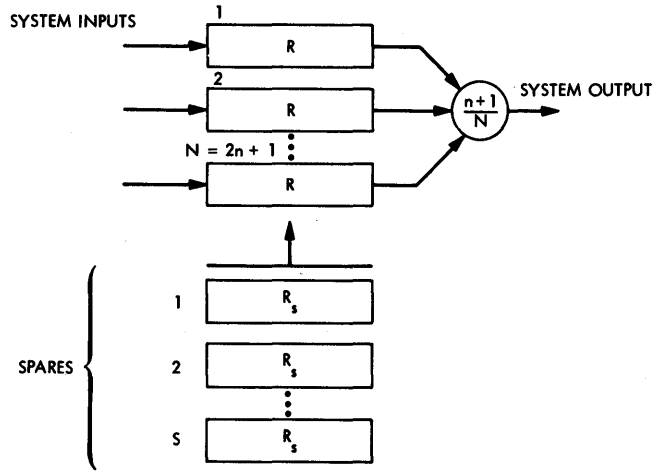


Figure 3—Hybrid (N, S) sytem concept

the outputs of each one of the active $2n + 1$ units. When a disagreement occurs, a signal is transmitted to the switching unit (SU), which replaces the unit that disagreed by switching it out and switching in one of the spares. If the spare were to fail in the dormant mode and was switched in on demand from the DD unit, the disagreement would still exist and the SU would again replace it by one of the spares. The Hybrid(N,S) system reduces to a simple NMR system when all the spares have been exhausted, and the whole system fails upon the exhaustion of all the spares and the failure of any $n + 1$ of the basic $2n + 1$ units. In the special case where $N = 3$ the Hybrid(N,S) system reduces to a Hybrid(3,S) system.[10] In the case of zero spares the Hybrid(3,S) system then reduces to Hybrid(3,0) which is the basic TMR system.

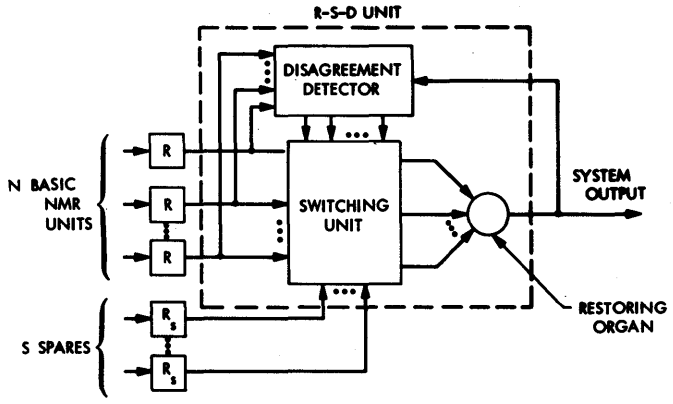


Figure 4—Hybrid (N, S) system block diagram

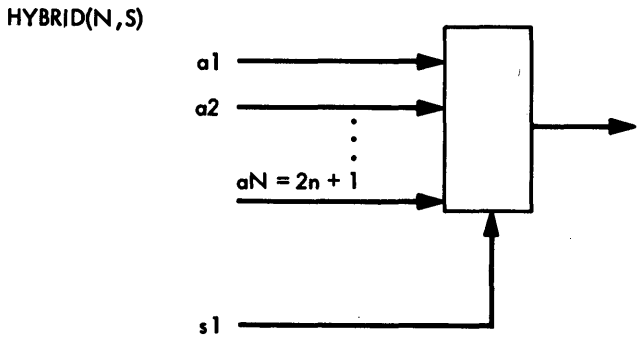
The Hybrid(N,S) system concept has been considered by other researchers from the architectural standpoint.[11,12] A derivation[13] of the reliability equation when dormancy of the S spare units is not considered (i.e., when all the $S + 3$ units in the system are considered to have identical failure rates) yields

$$R(3, S) = 1 - (1 - R)^{S+2}[1 + (R) \cdot (S + 2)]$$

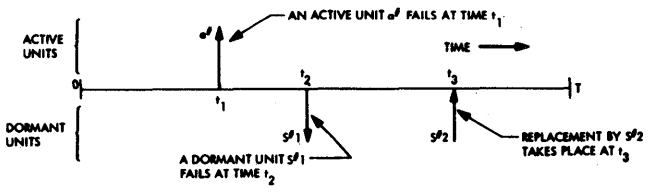
which is simply the probability that at least any two of the total $S + 3$ units survive the mission duration, when assumption is made that the majority organ and associated detection and switching logic are fail-proof.

DERIVATION OF $R(N,S)$, THE CHARACTERISTIC RELIABILITY EQUATION OF THE HYBRID(N,S) SYSTEM

First an expression for the reliability of Hybrid(N,1) system (i.e., $S = 1$) will now be derived. Let the N basic units be designated as $a1$, $a2$, $\cdots aN$, and the spare as $s1$, as follows:



Three cases may be distinguished which yield the success of the system for any mission time T. These three cases are illustrated by means of the line drawings shown in Figure B, Figure C, and Figure D. The notation of these descriptive drawings is explained in Figure A.



The nomenclature in Figure A is the following. The horizontal line represents the time axis from the start of the mission (time = 0) to the end of the mission

(time = T). The region above the lines is the domain of the active units (massively redundant) while the region below the line is the domain of the dormant units (selectively redundant). Arrows leaving the line represent failure of a unit. The direction of the arrow leaving the line towards the active or the dormant domain indicates failure of an active or dormant unit respectively. An arrow going towards the line indicates a replacement action where a dormant unit replaces a failed active unit, thus in Figure A t_3 would equal t_1 since the failure of an active unit demands a replacement from the spare bank.

Case (*i*). All units survive mission time T:

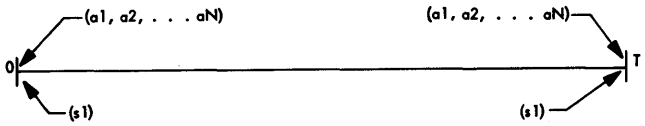
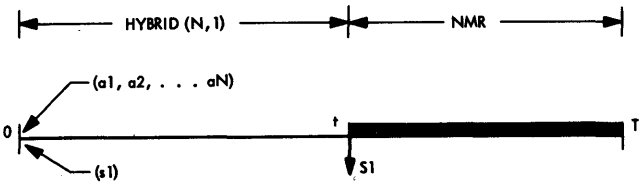


Figure B shows that the active units ($a1$, $a2$, $\cdots$ aN) which were good at time = 0 are still good at time = T and likewise for the dormant unit $s1$. This event has the probability $R^N \cdot R_s$.

Case (*ii*). The spare unit is the first unit to fail:



At some time t ($0 \leq t \leq T$) the spare unit $s1$ fails, leaving the system in basic NMR, i.e., Hybrid(N,0), for the unelapsed time $[T - t]$. The probability of this event is

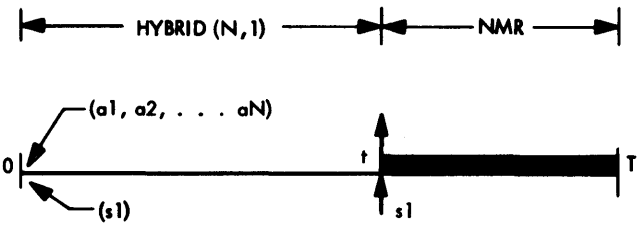
$$\int_0^T e^{-N\lambda t} \cdot \mu e^{-\mu t} \cdot R(N, 0)[T - t]\, dt$$

where

$$R(N, 0)[T - t] = \sum_{i=0}^{n} \binom{N}{i} \cdot (1 - R[T - t])^i \cdot R^{N-i}[T - t]$$

which is the reliability of the basic NMR system for a mission time $[T - t]$.

Case (*iii*). An active unit fails before the spare:



At some time t one of the basic N units fails and is replaced by the spare $s1$, thus leaving the system in basic NMR for the rest of the time $[T - t]$. The probability of this event is:

$$N \int_0^T e^{-\mu t} \cdot \lambda e^{-\lambda t} \cdot e^{-(N-1)\lambda t} \cdot R(N, 0)[T - t]\, dt$$

Summing the above three cases yields

$$R(N, 1)[T] = R^N[T]R_s[T] + (N\lambda + \mu) \int_0^T e^{-(N\lambda+\mu)t} \cdot R(N, 0)[T - t] \cdot dt \quad (3)$$

Similarly it may be shown that for the case of two spares

$$R(N, 2)[T] = R^N[T]R_s^2[T] + (N\lambda + 2\mu) \int_0^T e^{-(N\lambda+2\mu)t} \cdot R(N, 1)[T - t] \cdot dt \quad (4)$$

and, in general, for S spares

$$R(N, S)[T] = R^N[T]R_s^S[T] + (N\lambda + S\mu) \int_0^T e^{-(N\lambda+S\mu)t} \cdot R(N, S - 1)[T - t] \cdot dt \quad (5)$$

which may be rewritten by letting $\tau = T - t$ as

$$= R^N[T]R_s^S[T] \cdot \left\{1 + (N\lambda + S\mu) \int_0^T e^{(N\lambda+S\mu)\tau} \cdot R(N, S - 1)[\tau] \cdot d\tau\right\} \quad (6)$$

The recursive integral equation for the case of one spare ($S = 1$) has the solution

$$R(N, 1)[T] = R^N R_s \left[1 + (NK + 1) \sum_{i=0}^{n} \binom{N}{i} \cdot \sum_{l=0}^{i} \binom{i}{l} \frac{(-1)^{i-l}}{(Kl + 1)} \left(\frac{1}{R_s R^l} - 1\right)\right] \quad (7)$$

and the general solution for $(S > 1)$ is given by

$$R(N, S)[T] = R^N R_s^S \left[1 + \sum_{j=0}^{S-2} \binom{NK+S}{j+1} \cdot \left(\frac{1}{R_s} - 1\right)^{j+1} + \sum_{i=0}^{n} \binom{N}{i} \binom{NK+S}{S} \cdot \sum_{l=0}^{i} \frac{\binom{i}{l}(-1)^{i-l}}{\binom{Kl+S}{S}} \left\{ \left(\frac{1}{R_s^S R^l} - 1\right) - \sum_{j=0}^{S-2} \binom{Kl+S}{j+1} \cdot \left(\frac{1}{R_s} - 1\right)^{j+1} \right\} \right] \quad (8)$$

where $K = \lambda/\mu$; $\mu \leq \lambda$ and $1 \leq K < \infty$.

For the special situation of non-failing spares, we have $K = \infty$, (i.e., $\mu = 0$) and the solutions (7) and (8) reduce to:

(i) for $S = 1$

$$R(N, 1)[T] = R^N \left\{ 1 + \lambda NT(-1)^n \binom{2n}{n} + N \sum_{i=1}^{n} \binom{N}{i} \sum_{j=1}^{i} \binom{i}{j} \frac{(-1)^{i-j}}{j} \left(\frac{1}{R^j} - 1\right) \right\} \quad (7a)$$

(ii) for $S > 1$

$$R(N, S)[T] = R^N \left\{ \sum_{i=0}^{S-1} \frac{(N\lambda T)^i}{i!} + \frac{(N\lambda T)^s (-1)^N}{s!} \cdot \binom{2n}{n} + N^s \sum_{i=1}^{n} \binom{N}{i} \sum_{j=1}^{i} \binom{i}{j} (-1)^{i-j} \cdot \left[\frac{1}{j^s} \left(\frac{1}{R^j} - 1\right) - \sum_{l=1}^{S-1} \frac{(\lambda T)^l}{l!\, j^{s-l}} \right] \right\} \quad (8a)$$

The proof that equations (7) and (8) are the solutions to the recursive integral equation (6) may be verified by inserting them on the righthand side of (6) with parameter equal to $S - 1$. The meanings of all symbols in the above equations are summarized in Table 1.

In the derivation of the above equations it was assumed that the restoring organ, the switching unit, and the disagreement detector (jointly referred to as the R–S–D unit) are fail-proof. In order to incorporate the reliability of these units, they may be assigned a lumped parameter R_v, reflecting their reliability; and with the simplifying assumption that the R–S–D unit has a series reliability relative to the ideal Hybrid (N, S) configuration, the term R_v may be used as a product term to directly modify the reliability equations derived here.

DISCUSSION OF THE MODEL BEHAVIOR

The application of redundancy in general does not necessarily guarantee improvement in reliability. This is especially evident from the characteristic reliability curves of the simple NMR system as shown in Figure 2. It is to be noted that if R (the reliability of the non-redundant unit) is less than 0.5 (i.e., $\lambda T > 0.697$) then the system is worse off with redundancy. Furthermore the application of higher orders of redundancy (larger value of N) makes the system progressively worse. Also one of the characteristics of such a system is that the cross-over point where the redundant system reliability is equal to the non-redundant system reliability does not vary with the order of redundancy N. It sets a large lower bound on the reliability of the original system amenable to improvement by the application of the basic NMR form of redundancy technique.

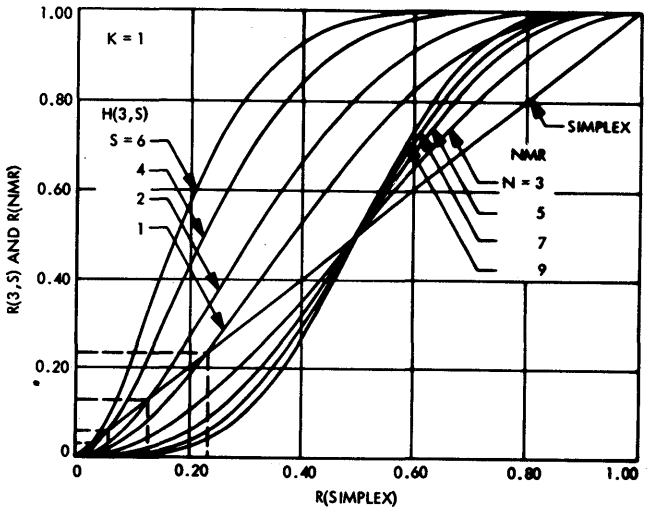


Figure 5—Comparative reliability curves of H (3, S), NMR, and simplex systems

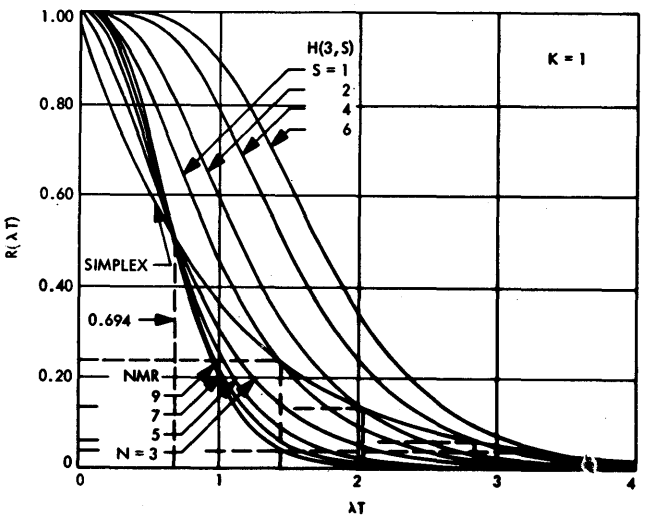


Figure 6—Reliability comparison of a H (3, S) and NMR systems vs normalized time λT

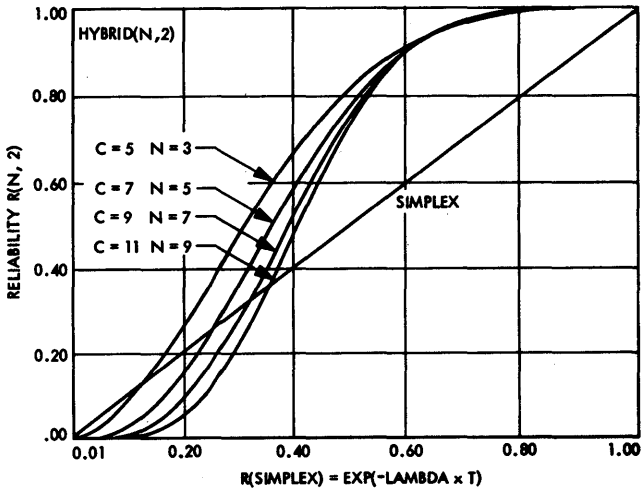


Figure 8—Reliability R (N, 2) vs R (Simplex)

The effects of hybridization (i.e., the addition of standby spares) on the NMR system with the replacement form of redundancy as analytically expressed by equation (8) are shown graphically in Figure 5 through Figure 10. The reliability of the Hybrid(3,S) system for the case of $N = 3$, $K = 1$ and with several values of S (the number of spares) is shown in Figure 5 and Figure 6. Also alongside for comparative purposes the reliability curves of the NMR system and the non-redundant system are also shown. In Figure 7 and Figure 8 are shown the reliability of the Hybrid(N,S) system versus the reliability R of the non-redundant unit for several values of N. They illustrate the effect of the variation of the order of redundancy N in the NMR core. In Figure 9 and Figure 10 are shown reliability curves for $K = 1$ and $K = 10$ respectively for various values of the number of spares S.

The improvement in reliability of the Hybrid(N,S) system over the NMR system is readily seen from the curves. It is to be noted that the well-known crossover point, which in NMR systems occurs at a reliability of 0.5 is significantly reduced in the Hybrid(N,S) system. With $N = 3$ and $S = 1$ the crossover point occurs at $R = 0.233$ for the value of $K = 1$, and rapidly diminishes with higher allocation of the number of spares ($S > 1$). The shift in the crossover point is also

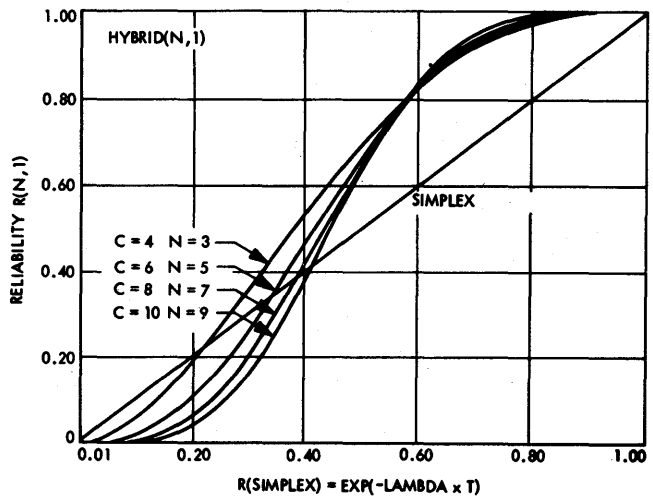


Figure 7—Reliability R (N, 1) vs R (Simplex)

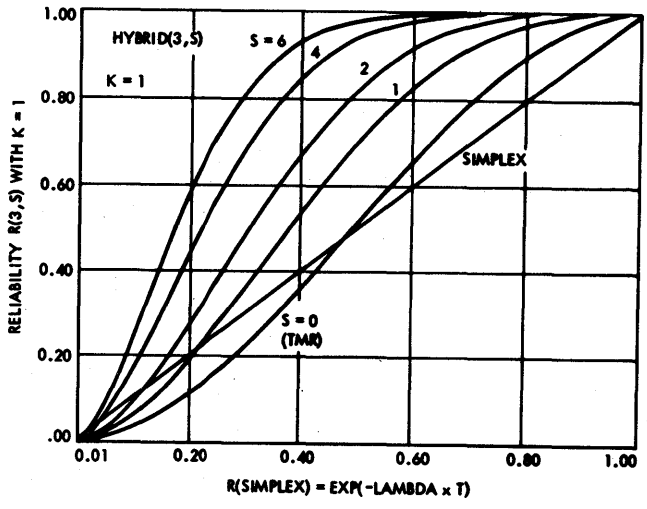


Figure 9—R (3, S) vs R (Simplex) with $K = 1$

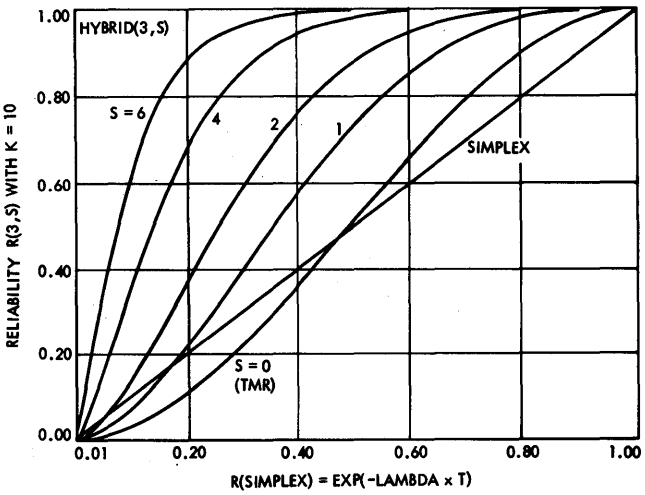


Figure 10—R (3. S) vs R (Simplex) with $K = 10$

sensitive to variations in the value of K. The effect of changes in values of K on the system reliability and the shifts of the crossover point become very slight when K exceeds the value of 10.

The decision as to how to allocate redundancy for a given total number of units $C = N + S$, where N is the number of active redundant units in the NMR core and S is the number of standby spares, is resolved by the curves shown in Figure 7 and Figure 8. Since N is always an odd number it follows that if C is odd then S is even and vice versa. The possible allocation policies are then as tabulated below.

With one spare, $S = 1$, as shown in Figure 7 the improvement in reliability in going to higher order N of active redundancy is restricted to the range $0.58 < R < 1$. When the number of spares is increased to two, $S = 2$, with N as the variable, the range of improved reliability is further restricted to $0.65 < R < 1$. Also, within this shrinking range (as a function of increased S), the improvement in reliability due to larger values of N also tends to become less significant. This indicates that the order of massive redundancy N should be kept at a minimum in the NMR core (i.e., $N = 3$). Maximum redundancy should be inserted in the spares bank, thus in practical implementation N should equal three, with S as variable to suit the desired level of mission reliability.

TABLE II

ALL POSSIBLE ALLOCATION POLICIES OF C

C_0 is odd		C_e is even	
$N = 3$	$S = C_0 - 3$	$N = 3$	$S = C_e - 3$
$N = 5$	$S = C_0 - 5$	$N = 5$	$S = C_e - 5$
.	.	.	.
.	.	.	.
.	.	.	.
$N = N$	$S = C_0 - N$	$N = N$	$S = C_e - N$
.	.	.	.
.	.	.	.
.	.	.	.
$N = C_0 - 2$	$S = 2$	$N = C_e - 3$	$S = 3$
$N = C_0$	$S = 0$	$N = C_e - 1$	$S = 1$

Hardware utilization and hence cost is another major advantage of the Hybrid(N,S) redundant system. Efficient hardware utilization over comparable NMR systems is due to the fact that for an equal number of total N units the NMR system will tolerate failures of only $(N - 1)/2$ units whereas the Hybrid $(3,S)$ system will tolerate as many as $N - 2$ failures. Thus when an NMR system fails it leaves behind n good units while in the Hybrid$(3,S)$ system only one good unit remains upon system failure. In general the Hybrid(N,S) system upon exhaustion of all spares and subsequent failure of the system leaves $(N - 1)/2$ good operating units which is a minimum when $N = 3$. Thus another argument for keeping the parameter N confined to the value three in Hybrid(N,S) system is this of efficient hardware utilization.

ACKNOWLEDGMENTS

The authors wish to thank William F. Scott, John J. Wedel, and George R. Hansen of the Flight Computers and Sequencers Section of the Astrionics Division of the Jet Propulsion Laboratory for their constant encouragement and for providing the atmosphere conducive to this research. Thanks are also due to Prof. Leonard Kleinrock of the University of California, Los Angeles for his advice on the subject of queueing theory; the notation used herein to describe the dynamics of replacement has been adapted from similar notation used to describe the behavior of queues.

This paper represents in part research which has been carried out at the Jet Propulsion Laboratory under NASA Contract NAS7-100.

REFERENCES

1 S AUROBINDO
Savitri—A legend and a symbol
Sri Aurobindo International University Centre Collection Vol II Pondicherry India 1954

2 J VON NEUMANN
Probabalistic logics and the synthesis of reliable organisms from unreliable components
In Automata Studies p 43–98 Princeton University Press Princeton New Jersey 1956

3 E F MOORE C E SHANNON
Reliable circuits using less reliable relays
J of the Franklin Institute Vol 262 Pt I pp 191–208 and Vol 262 Pt II p 281–297 1956

4 J E ANDERSON F J MACRI
Multiple redundancy application in a computer
Proc 1967 Annual Symposium on Reliability p 553–562 Washington 1967

5 A A AVIZIENIS
Design of fault-tolerant computers
AFIPS Conference Procedings Vol 31 p 733–743 1967

6 A A AVIZIENIS F P MATHUR D RENNELS J ROHR
Automatic maintenance of aerospace computers and spacecraft information and control systems
Proc of the AIAA Aerospace Computer Systems Conference Paper 69–966
Los Angeles September 8–10 1969

7 J K KNOX-SEITH
Improving the reliability of digital systems by redundancy and restoring organs
PhD thesis Electrical Engineering Stanford University August 1964

8 R F DRENICK
The failure law of complex equipment
J Soc Ind Appl Math Vol 8 No 4 p 680–690 December 1960

9 F P MATHUR
Reliability study of fault-tolerant computers
In Supporting Research and Advanced Development Space Programs Summary 37–58 Vol III p 106–113 Jet Propulsion Laboratory Pasadena California August 31 1969

10 F P MATHUR
Reliability modeling and analysis of a dynamic TMR system utilizing standby spares
Proc of the Seventh Annual Allerton Conference on Circuit and Systems October 20–22 1969

11 J GOLDBERG K N LEVITT R A SHORT
Techniques for the realization of ultrareliable spaceborne computers
Final Report Phase I Project 5580 Stanford Research Institute Menlo Park California October 1967

12 J GOLDBERG M W GREEN K N LEVITT H S STONE
Techniques for the realization of ultrareliable spaceborne computers
Interim Scientific Report 2 Project 5580 Stanford Research Institute Menlo Park California October 1967

13 J P ROTH W G BOURICIUS W C CARTER P R SCHNEIDER
Phase II of an architectural study for a self-repairing computer
International Business Machines Corporation Report SAMSO TR-67-106 November 1967

The architecture of a large associative processor

by GERALD JOHN LIPOVSKI

University of Florida
Gainesville, Florida

INTRODUCTION

This paper will describe features of architectural significance to the segmentability of a processor; it is not intended to be a detailed description of a processor for Information Storage and Retrieval. We regret that the incorporation of some features cannot be defended here because of the length of this paper. They are presented in a report.[16] We first state the types of problems to be processed. This will lead to the overall organization of the processor. In Information Storage and Retrieval, a processor should have the capability to store data which is formatted as ordered sets or unordered sets, and to retrieve all such sets having a specified subset. An *unordered set search* for a given subset S retrieves all sets containing S. An *ordered set search* for a given ordered subset S retrieves all ordered sets containing S. A *string search* for a given string S retrieves all ordered sets (strings) having a substring S. For example, if $S = (s_1, s_2, s_3)$ and $S_1 = (s_1, a, s_2, s_3)$, $S_2 = (a, b, s_1)$, $s_2, s_3, c, d)$, $S_3 = (s_2, s_1, s_3)$ and $S_4 = (s_1, a, b, s_2)$. Then an unordered set search for S would retrieve S_1, S_2, S_3, an ordered set search for S would retrieve S_1 and S_2, and a string search for S would retrieve S_2.

By storing ordered sets and unordered sets and allowing pointers to another set to be stored as elements of a set, one can store data formatted as a colored relational graph.[22] By permitting ordered sets or unordered sets to be elements of another ordered set or unordered set, one can store data formatted as one of several types of trees. Thus, the capability to store ordered and unordered sets is sufficient to store most useful types of data. The retrieval capability of this processor extends beyond set, ordered set, and string searches, but this feature will not be discussed.

A class of processors which are well suited to the problems discussed above are those having a linear array of associative memory cells (*Linear array processors*). Each cell has a fixed size *word* of memory, W, and a comparator. All cells simultaneously receive a word C and a mask M, which are broadcast in the channel; normally, a cell is said to match if $1 = \wedge/(C = W) \vee M$ (Iverson notation). A set is generally stored in a collection of contiguous cells (*aggregate*) with one element of the set in each cell. Contiguous cells are consecutively numbered in Figure 1. (The numbers are for descriptive purposes in this paper, they are not addresses.) One or more rails between cells are used to combine the results of matches from various cells in the aggregate to detect the existence of a given ordered subset, or substring, in the ordered set stored in the aggregate, or the existence of a given subset in the set stored in the aggregate.

A linear array processor was first used by Lee and Paull for string searches.[14] Gains and Lee[9] found it useful for ordered set searches, and Savitt et al.[23] found it useful for unordered set searches. Sturman[28] showed it possible to store and broadcast instructions from these associative memory cells, and therefore dispense with the need for a central processing unit, and Smathers[25] has added several practical improvements to this iterative processor model. These approaches were consolidated into an iterative processor designed for set, ordered set, and substring searches.[16]

For information retrieval the processor should be large to efficiently handle large data sets, and it should have the capability to be loaded and unloaded quickly. Since the same cell can be made to store data or broadcast an instruction, it is clear that an iterative processor based on any of the previously discussed linear array processors can be *segmented* into several independently acting collections of cells, each executing a different program or different parts of the same program. We have found that the capability to broadcast instructions and segment a processor increases the "cost" (number of gates) in a basic associative memory cell by 18% in a possible realization of a processor which was studied. Therefore, if 36% of all programs run on the processor can be executed in pairs of segments simultaneously in the processor, the capability to segment

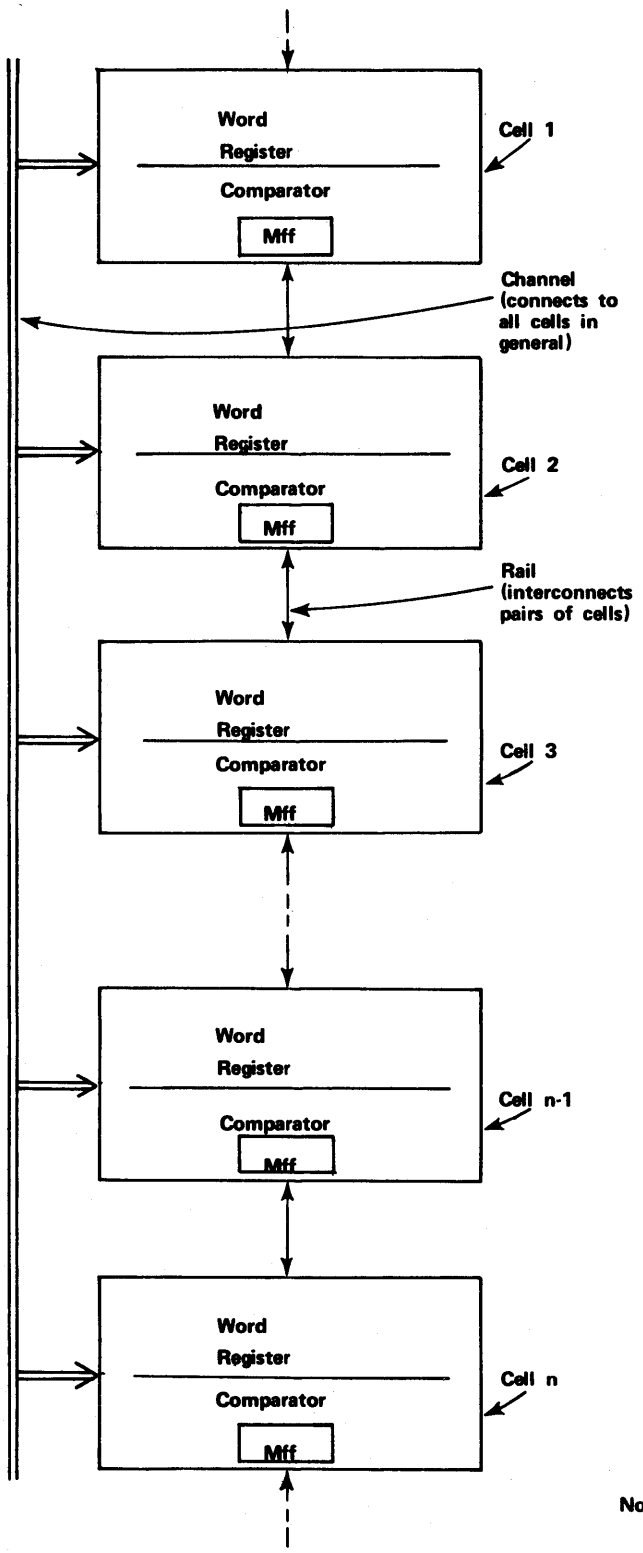


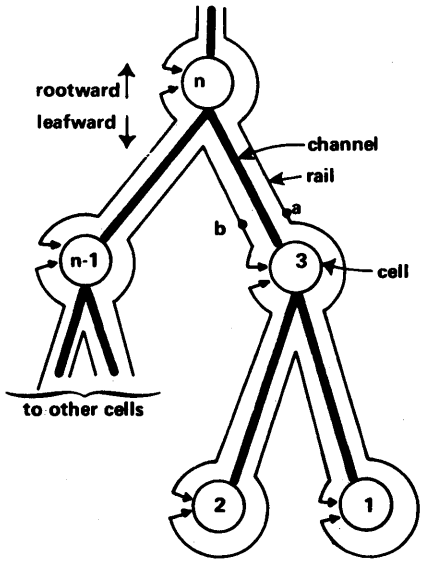
Figure 1—A linear array processor

a processor is economically justified. This is expected to be the case in large processors. A high degree of parallel programming for subprograms of the same program is made available, and each segment can be connected to a different I/O device for parallel, more efficient, loading and unloading of information in a segmentable processor.

THE TREE CHANNEL PROCESSOR

A linear array processor (Figure 1) suffers from excessive propagation delay and a succeptability to faults, especially where a cell output amplifier is stuck-on-one. Propagation delay on the rail, where a signal may have to propagate through many cells in one clock period, is especially slow. Figure 2 shows a better connection scheme (*tree channel processor*).

The word, C, normally is broadcast to all cells "simultaneously" via the channel in the tree branches shown, and rails are used to communicate the results of



Note: Cell numbers are for descriptive purposes in this report; they are not addres
Cell numbering is determined by relative position of the
cell on the rail. The root cell always has the largest number.

Figure 2—A dubtree of a tree channel processor

searches in aggregates just as in the previous connection scheme. This processor has two rails, each connected as the rail in Figure 2. Note that the rail connects consecutively numbered cells in Figure 2 just as in Figure 1. (Cell i is said to be *above* cell j, $j > i$ and *below* cell k, $k < i$.) However, if a signal at point "a" on the rail must propagate to and through cells 1, 2, and 3 to point "b" and beyond in Figure 2, it will "short cut" directly through cell 3. This decreases the delay time. The maximum propagation delay time through gates and through transmission lines in the channel or rails determines the clock rate of the processor. For a processor having n cells, it grows like a $\log(n) + b(n)^{\log_7 2}$ in a 7-way homogenous tree,* which is conveniently realizeable in three dimensions.

* Iverson Notation[11]

The tree structure is economically segmented. In Figure 3, by setting a flip-flop in, say, cell 3, the channel can be disconnected between cell 3 and cell 7 (*delimit the channel at cell 3*). This forms a subtree, cells 1, 2, and 3. In each subtree both rails are effectively reconnected between pairs of cells to provide a "linear array," as in Figure 2 or 7. Within each subtree, in one clock time unit, some cell broadcasts its word into the channel to all cells in that subtree. Each cell amplifies the channel signal and propagates it as soon as possible. All cells obey this instruction in the same time unit. Each subtree operates independently and simultaneously. Because these subtrees define the extent of broadcast of instructions, they are called *instruction domains.* (ID's)

While ID's normally act independently, they can issue instructions to delimit the channel or reconnect it (*processor management*). Input output is provided at various leaves of the tree, as in Figure 3. An ID, for example, "A" in Figure 3, gains control of an I/O channel channel below cell 8 by *merging* with ID's "B" and "C". It does this by changing the flip-flops that delimited the channel in cells 3 and 8. It is clearly possible to simultaneously connect ID "F" to the other I/O unit and load data into cells in "F" while "A" is also being loaded.

This architecture has two basic drawbacks. The placement of I/O channels is fixed; for example, it is inconvenient for ID "A" to utilize the I/O channel below cell 17 (Figure 3), and impossible, while this I/O transaction is taking place, for ID "F" to gain access to the I/O channel below cell 8. Further, as we show in a later section, for each I/O transaction, programs in all ID's are temporarily stopped while some ID's are merged in order to connect an I/O channel to the ID requesting it. Later, programs in all ID's must again be stopped to restore the previous arrangement of ID's. In a large processor, the "overhead" to begin and terminate each I/O will be high. To circumvent these difficulties, a switching network (SW-structure) is used at the root of the processor tree. The resulting architecture has some properties of a computer network; it will be discussed later.

ESSENTIAL CHARACTERISTICS OF THE CELL

A description of the cell will be incompletely given for two reasons. Firstly, a complete discussion of the instruction set for the processor which was studied would be too long and would detract from the study of the segmentation of a processor. Secondly, the results obtained here apply to various iterative processor architectures in which instructions are broadcast in a channel.

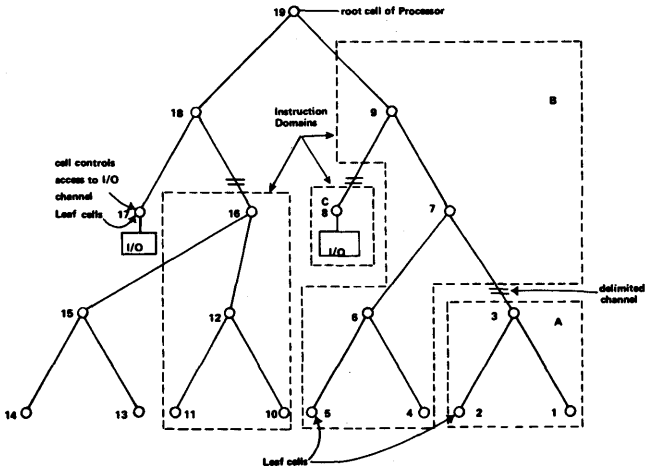


Figure 3—A processor with I/O

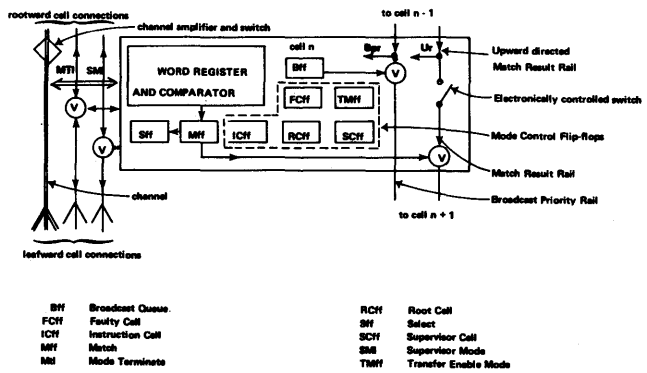


Figure 4—The essential cell

TABLE I—Cell, Instruction Domain, and Response Modes

Cell modes	Instruction domain modes: Assignment SM = 1, TM = 0			Supervisor 1	Transfer Enable 0	Run 0
	RC	SC	IC			
Independent root	1	1	0	Data	B-delimit	B-delimit
Dependent root	1	0	0	Data	R-delimit	R-delimit
R-instruction	0	0	1	Passive	Passive	Instruction
S-instruction	0	1	1	Instruction	Passive	Passive
R-data	0	0	0	Passive	Transfer	Data
S-data	0	1	0	Data	Passive	Passive
Faulty	FC = 1			Passive	Passive	Passive

The essential elements of the cell are shown in Figure 4. The flip-flops, FCff, TMff, ICff, RCff, and SCff, and the lines MTr and SMr, are used to control the processor as we shall now describe. In the following discussion of modes and mode changes, for the sake of concreteness and clarity, a tentative assignment of states, or modes, to these flip-flops and lines will be made.

The tree channel processor possesses *instruction domain modes*. These modes are indicated in each cell by SMℓ and TMff (Figure 4 and Table I). In a given ID, all cells are in the same instruction domain mode. The *run mode* enables the normal execution of instructions for information retrieval. The *transfer enable mode* provides for efficient loading and unloading of words in the word register of cells by means of the channel. The *supervisor mode* enables channel delimiting cells to be set up or changed. The operation of these three modes will be explained in the next three sections.

All cells are essentially identical in construction and capability in this iterative processor. Each cell, however, has a *cell mode* fixed by RCff, SCff, ICff, (Figure 4 and Table I) to assign to it a specific modus operandi. For a given instruction domain mode and a given cell mode, a cell has a *response mode* (Table I). The *data mode*, or data response mode, permits a cell to store data and interpret instructions that search or write in data cells. The *instruction mode*, or instruction response mode, permits a cell to issue instructions. In the *passive mode* a cell will not issue instructions, it cannot be written in, and none of its control flip-flops can be changed directly by instructions in the channel. Rails to the cell immediately above it are connected to the cell immediately below it, and the cell itself does not broadcast a "one" signal on either rail. Cells detected to be faulty can have FCff set; they then become permanently passive. Both the *B-delimiting* (bidirectorial delimiting) and R-delimiting (rootward delimiting) modes are similar to the passive mode except that they cause a cell to delimit the channel. A B-delimiting cell completely separates the channels of two ID's, while an R-delimiting cell permits leaf-ward broadcasting through it but delimits rootward broadcasting through it. In either case, the rail is reconnected to connect consecutively numbered cells in each instruction domains set up as in Figure 2. For example, in Figure 3, if cell 3 is B-delimiting, the instruction in the channel of ID "*A*" must be broadcast from some cell of ID "*A*." On the other hand, if cell 3 is R-delimiting, the instruction in the channel of ID "*A*" is the bit-wise OR of the instruction broadcast by a cell in ID "*B*" and the instruction broadcasting a cell in ID "*A*." This type of connection is particularly useful in diagnoising cells.

OPERATION IN THE RUN MODE

In the run mode, *R-data* cells are in the data mode and *R-instruction* cells are in the instruction mode. The generation of instructions in a segmentable processor poses some problems. It is possible, of course, to merge two ID's, both broadcasting their own sequences of instructions, into one ID, in which only one sequence of instructions is broadcast. The machinery for selecting a cell in an ID to broadcast must be able to automatically resolve which sequence broadcasts and when. The Z-propagating rail[26] does not satisfy this requirement.

An instruction is any command that a programmer can effect by causing some cell to broadcast its stored word. In each ID, at each clock pulse, one instruction is read into the channel and is broadcast to all cells in the ID. The mechanism for selecting this broadcasting cell is the broadcast queue flip-flop, Bff, and the broadcast priority rail, BPr, in a conventional priority-determining

circuit. The set of data or instruction cells with $B = 1$ constitute the *broadcast queue*. By means of BPr, which broadcasts a "one" signal from cell i, if $B = 1$ in cell i, to BPℓ in all cells j, $j > i$, a *B-prior* cell will be selected; it will have $B = 1$, BP $= 0$. This cell will broadcast its word in the channel and reset Bff. Thus, at the next clock pulse, another cell becomes B-prior and broadcasts in the channel.

For example, if $B = 1$ in cells 2, 3, 4, and 6 in Figure 5, cell 2 would set BP $= 1$ in cells 3–7; cell 2 only would broadcast. When cell 2 resets Bff, cell 3 broadcasts, then cell 4, and cell 6 in turn.

Cells in the data response mode are assumed to obey at least the following instructions. A Match instruction resets Mff in data cells, then sets Mff if the comparator detects a match between the contents of the channel and the word stored in the cell. The match result rail has a switch controlled by some flip-flop in the cell (not shown); by means of this rail, the value in Mff in a cell is broadcast so that string searches, at least, can be carried out. We also require an instruction or program for selecting the one cell i with $M = 1$ and smallest integer i for which $M = 1$ (*priority instruction*). A write instruction is assumed, to change the word in a cell. Several processors have these basic instructions.[14,9,22,16] We have included a flip-flop, Sff, in our cell for the transfer enable mode; we require some instruction to load Sff from Mff. Other instructions will be explicitly mentioned in the following sections.

Instruction mode cells can be made to broadcast their word if Bff is set. This setting is accomplished with the help of the match result rail (Figure 4). Broadcasting the JUMP instruction clears Bff in all instruction cells and loads the value, U, on the match result rail into Bff. Suppose, in Figure 5, that the switch in the match result

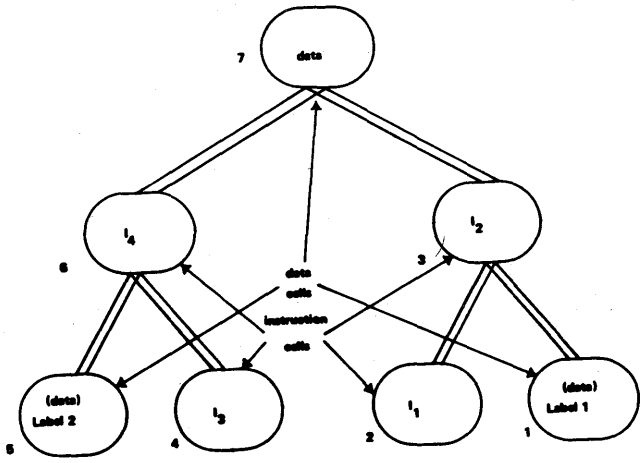


Figure 5—An ID with instructions

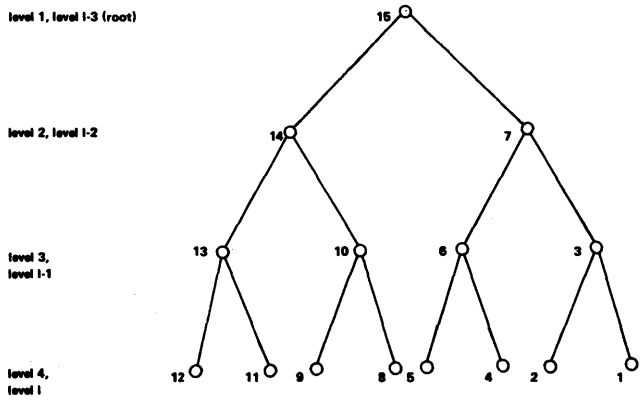


Figure 6—A tree

rail is open only in cell 7. If the most recent match instruction set Mff in cell 1, a data cell, then $U = 1$ in cells 2–7; a JUMP instruction sets $B = 1$ in cells 2, 3, 4, and 6 and clears Bff in any other instruction cell. Cell 2 broadcasts next. If the most recent match instruction set Mff in cell 5, then $U = 1$ in cells 6 and 7; a JUMP sets $B = 1$ in cell 6 only and clears Bff in cells 2–4 and any other instruction cells. This simple mechanism can be used in programming a variety of conditional jumps as well as unconditional jumps.

It will be shown that all cells are initially in the R-data mode. In this mode, an addressing scheme for locating cells to become channel delimiting is necessary to segment the processor in a predictable way. In Figure 3, for example, one must be able to locate cell 9 to set up instruction domain B. An address for each cell is neither practical nor desirable. Instead, only leaf cells, cells in level 4 in Figure 6 are marked.

An instruction, LOCATE LEAF, sets $M = 1$ in all these leaf cells and resets Mff in all other cells. Consider a 2-way homogenous tree in Figure 6. A cell n is in level $\ell - k$ if cell $n - k$ is a leaf cell and all cells m, $n - k < m \leq n$, are not leaf cells. For example, cell 7 is in level $\ell - 2$ since cells 7–0 $= 7$, 7–1 $= 6$ are not leaf cells, and cell 7–2 $= 5$ is a leaf cell. It is clearly possible to replace the match instruction with a LOCATE LEAF instruction in a string search in order to set Mff in all cells of level $\ell - k$. The exact mechanism for a specific instruction set depends on the way string searches are programmed with that instruction set. The priority instruction can then be used to "count down" cells in level $\ell - k$ to choose a specific cell.

Repeated use of this method can be used to locate any cell in the tree in a short time. For example, to locate cell 9 in Figure 6, we use the above method to locate cell 14, then in the subtree below cell 14, we use this method to locate cell 10, and in its subtree, we locate

TABLE II—Mode Changing Instructions

INSTRUCTION	ACTION
SET *IC*	In data cells where $U = 1$, set *ICff*
RESET *IC*	In instruction cells where $U = 1$, reset *ICff*
SET *SC*	In data cells where $M = 1$, set *SCff*
RESET *SC*	In data cells where $M = 1$, reset *SCff*
SET *RC*	In the supervisor mode, in data cells where $M = 1$, set *RCff*
RETSET *RC*	In the supervisor mode, in data cells where $M = 1$, reset *RCff*

cell 9. This method reduces the amount of "counting" using the priority instruction. The maximum number of program steps to locate any cell grows logarithmically with the number of cells in the processor. Once the cell has been located, a unique word can be stored in its word register so that it can later be found with a simple match instruction.

Since all cells are initially in the R-data mode, some instructions are required to change cell modes. To change cells found by the technique just described to channel delimiting cells, they are first changed into S-data cells in the run mode, and then to root cells in the supervisor mode (SM = 1). One must be able to convert R-data cells to R-instruction cells, and vice versa to compile and then execute instructions. These mode changes are carried out with instructions given in Table II. Note that ICff is reset or set from the signal on the match result rail, similar to the JUMP instruction. Lastly, instructions are required to change instruction domain modes. These, the TRANSFER CALL and SUPERVISOR CALL, will be examined in turn.

OPERATION IN THE TRANSFER MODE

A *transfer* is the operation of broadcasting a word either from one R-data cell or from an input channel and writing that word in an R-data cell or sending it to an output channel. An efficient transfer mechanism is required to load large data bases into or out of a large processor. This is provided by the transfer enable mode, which is described in simplified terms below.

This mode is entered when the instruction, TRANSFER CALL, is broadcast. In the instruction domain, R-instruction cells become passive, and R-data cells become transfer cells. The contents of Sff which were obtained earlier from Mff are loaded into Bff in all R-data cells at this time. Whatever sequence of instructions was being broadcast from R-instruction cells is temporarily halted since these cells become passive, and the contents of the channel are not interpreted as instructions even though they would be legitimate instructions in the run mode.

The combination, Bff and BPr, is used to select words to broadcast, as in the run mode, and Mff and the match result rail, with all switches closed (Figure 4) are used to select a cell to be written in, in a similar way. If any transfer cell has $B = 1$ the transfer cell with $B = 1$, $BP = 0$ broadcasts its word into the channel and resets Bff. If any transfer cell has $M = 1$, the transfer cell with $M = 1$ and $U = 0$ writes this word in the channel into its register and resets Mff.

If an I/O channel is connected to the instruction domain in the transfer enable mode, words broadcast from cells will be sent to the I/O channel. To output some words, we set $S = 1$ in cells containing these words, we reset Mff in all R-data cells so these words are not written in other cells, and we merge ID's to include the I/O channel. Some transfer cells above these cells contain channel comments in their word register; we set $S = 1$ in these cells too. Then in the transfer mode, the channel commands are first broadcast to select an output device and format, and the words to be output are broadcast. To input some words, we use the transfer mode twice. The first time, we output channel commands to select an input device and format. Then, we reset Sff in all R-data cells and set $M = 1$ in cells to be written in. When we re-enter the transfer mode, words are read from the input device until an end-of-file is encountered. While the I/O channel broadcasts words into the channel, it broadcasts a "one" signal in BPℓ to the root cell of the ID (See Figure 7). Mff and the match result rail selects one cell each time to write the word broadcast from the input channel.

The transfer enable mode is terminated when neither the I/O channel nor any cell is broadcasting. In this condition, $BP = 0$ in the root cell of the ID, and that cell broadcast $MT = 1$ to all cells in the ID. (See Figure 7) These cells reset TMff. The ID returns to the

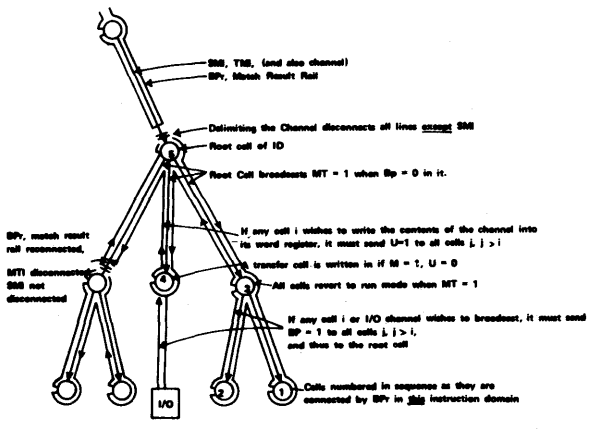


Figure 7—Control lines in the transfer enable mode

run mode, and R-instruction cells, no longer passive, continue broadcasting the instruction sequence exactly where they left off when the transfer enable mode was entered.

OPERATION IN THE SUPERVISOR MODE

The supervisor mode is provided for processor management. In it, dependent and independent root cells are set up or changed, thereby, ID's are changed. (See Figure 8) These root cells and S-data cells are in the data mode, and can be searched, written in, or changed. S-instruction cells, which are in the instruction mode, are created only when the supervisor mode is entered.

If some ID in the processor wishes to enter the supervisor mode to change the segmentation structure, it broadcasts the instruction, SUPERVISOR CALL. This sets SCff in all R-instruction cells in that ID. These become S-instruction cells. Whenever any S-instruction cell exists, it continuously broadcasts a "one" signal on the supervisor mode line, SMℓ, to all cells in the processor (Figure 7). This signal causes all cells, and thus all ID's, to simultaneously enter the supervisor mode, and remain in it as long as $SM = 1$ (Figure 8). All R-data cells, and all R-instruction cells in ID's which did not broadcast SUPERVISOR CALL, become passive. Therefore, if some ID were interrupted while executing some program because some other ID broadcast a SUPERVISOR CALL, all information pertinent to that program is "frozen" in passive cells. If that ID returns to the run mode, it will continue executing the program where it left off. This is also true if the ID were in the transfer enable mode when the supervisor mode was entered. A programmer can therefore ignore the possible interruption of his program by an entry into the supervisor mode unless the ID in which he is programming is changed during the interruption.

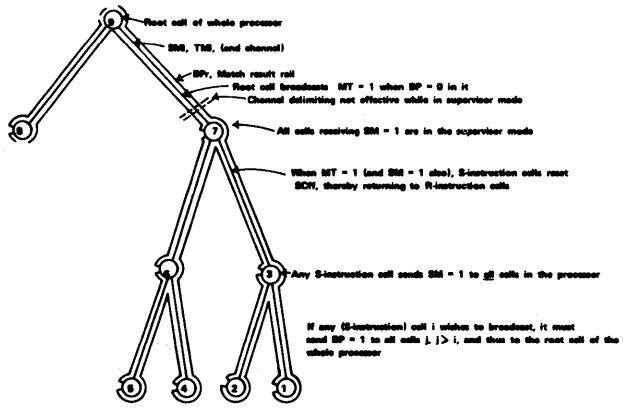


Figure 8—Control lines in the supervisor mode

In the supervisor mode, the instruction sequence is provided by S-instruction cells. Data cells are dependent and independent root cells, and S-data cells, all of which could not be changed while an ID was in the run mode. The entire processor appears to be a single instruction domain, although almost all cells are passive. All the search and write instructions that were available in the run mode are available in this mode to identify root cells or write in them, and the instructions, SET RC and RESET RC (Table II) can be used to set them up or change them.

It is possible for two ID's in the run mode to simultaneously broadcast SUPERVISOR CALL. The instruction sequence for the supervisor mode will be a mixture from the S-instruction cells in both ID's. Further, two ID's may wish to change the processor segmentation structure in contradictory ways at approximately the same time. To accommodate these possibilities, some software conventions are necessary. In one of these, a separate ID is designated the *executive*. It alone will keep an updated account of the processor structure. In the supervisor mode, it alone will change root cells. Other ID's that wish to change the structure will set flags in some R-data cell and change this cell to an S-data cell while the ID is still in the run mode. The executive will then periodically inspect these S-data cells when in the supervisor mode to determine what requests are impending.

When no more instructions are broadcast in the supervisor mode, $BP = 0$ in the root cell of the processor. (See Figure 8) This cell broadcasts $MT = 1$ to all cells. S-instruction cells reset SCff, when $MT = SM = 1$, thereby becoming R-instruction cells. Since S-instruction cells no longer exist, no "one" signal is broadcast on SMℓ to all cells. All ID's then return to the run mode or transfer mode that they were in just before they entered the supervisor mode.

We remark that some mechanism must exist to start a sequence of instructions broadcasting in a new ID when it is set up. One way is to let the word 00 . . . 0 in the channel be the code word for JUMP. Before the new ID is segmented away from the "parent" ID, the latter can broadcast a MATCH instruction to set $U = 1$ in selected R-instruction cells in what will be the new ID. After segmentation has been complete, the first instruction in the run mode in the channel will be JUMP because no cell will broadcast. This will cause a sequence of instructions to be broadcast from the selected cells in the new ID.

SOME REMARKS ON THE TREE STRUCTURE

The architecture of the tree array processor has two unrelated properties that are, nevertheless, quite important. Some hints of these properties appeared earlier. We now discuss the capability of a tree array processor to test for and operate in the presence of faulty cells, and the propagation delay in a physical realization of a tree array processor.

Before initializing a processor or instruction domain, it is expedient to test it for faulty cells. The test will change the word stored in the word register. While the exact nature of these tests depends upon the hardware realization of the cell finally selected, we can sketch some points of architectural significance here.

First, by means of an extra line, we force all cells to become dependent root cells ($FC = SC = IC = 0$, $RC = 1$). In this situation, an output amplifier stuck-on-one in a cell will not prevent correct testing of most other cells because the "one" signal it generates can travel only leaf-wards to cells in its subtree. For similar reasons, for a tree of ℓ levels, after ℓ cycles no cell will be broadcasting. All cells are then simultaneously diagnosed. Under the control of additional lines, the channel, word register and comparator are checked out by writing the same words in all cells and by matching for these words and for variations of them that would be caused by errors. Then all the rails are checked out. Cells found to be faulty have FC set in them. (We are assuming an idealized fault detection mechanism where the fault detection circuitry is itself free of faults.) If a channel amplifier is stuck-on-one, all cells leafward from it will be diagnosed faulty. If, for a given cell, all cells connected next to it in the leafward direction are faulty, the whole subtree below a cell is cut off by making the cell permanently B-delimiting and the cell itself sets FC. For example, in Figure 2, if cells 1 and 2 are faulty, then cell 3 is made permanently B-delimiting, and cells 1, 2, and 3 become inaccessible. The processor can run in the presence of the faulty cells.

The physical arrangement of cells in a processor will now be considered. The primary goal is to estimate the propagation delay time, and thus determine the clock rate to evaluate the cost of programming. A secondary goal is to find a spacial realization of the tree structure. In this section, we shall consider a good realization of a 7-way homogeneous tree. (A similar realization exists for 25-way or 50-way homogeneous trees. However, a tree fanout of seven permits electrical fanouts of rail amplifiers that are reasonable for integrated circuits.)

The following heuristic procedure generates a tree structure by beginning at the tree leaf cells. Put cells in the center and at seven of the corners of a cube. Mark the remaining corner "A." Link each corner to the center cell. This is a *leaf module.* In general, connect seven identical modules so that their "A" corners coincide, put a cell where these coincide, put a link from that cell to the free corner, corner "B," of the larger module just created (Figure 9). This procedure can be repeated by connecting seven identical modules of the kind just made so that their "B" corners coincide, and so on. Any 7-way homogeneous tree can be realized by repeating the above process.

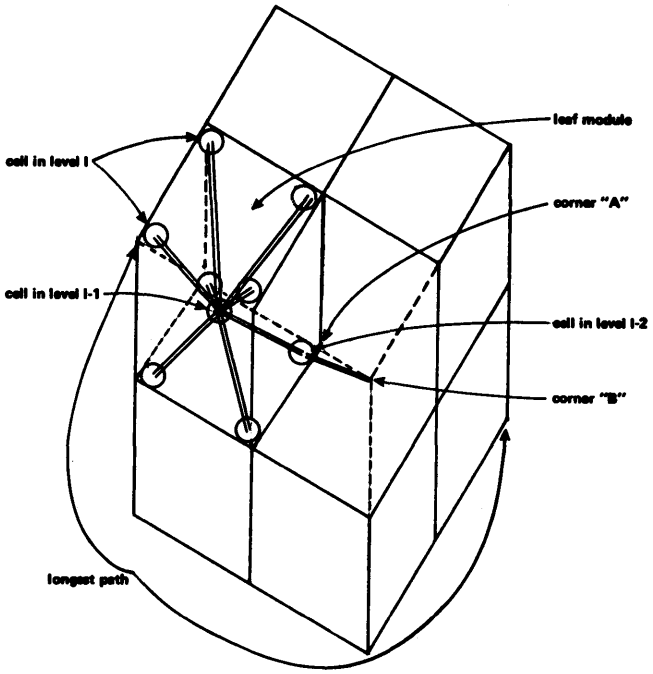


Figure 9—Connection of leaf modules

The clock rate for such a structure is determined by the propagation time in any line or rail through, at most, $2(L - 1)$ cells, each with bulk delay β due to gate delays in amplifiers, and by the transmission time of the pulse on links in the structure. The longest transmission path in this structure is on a straight line through opposite corners of the largest module (Figure 9). To find this length, suppose that a cube with edge ϵ will house a processor cell and provide enough room for a mechanical mole to test or replace the cell. The leaf module can be housed in a cube with edge 2ϵ. (To do this, the center cell of the leaf module is moved into the available space towards the "A" corner of it.) A tree of L levels can be put into a cube with edge $2^{(L-1)}\epsilon$. (Again, the center cell can be moved into unoccupied space in each module.) The longest transmission path is $\epsilon\sqrt{3}/2\ 2^L$. The clock period is therefore approximately $\sqrt{3}/2\ 2^L + 2\beta(L - 1)$ where α is the transmission delay of a pulse on a transmission line of length ϵ.

Consider an estimated propagation time of a typical processor. A 7-way homogeneous tree with 8 levels has 960,800 cells. If ϵ is three inches, if pulses propagate at 1ns./foot, β is 20 nanoseconds, and 100ns is required to decode the instruction in each cell, then the clock period of a processor with nearly one million cells is expected to be about 436 nsec.

THE SW-STRUCTURE

An earlier section presented a mechanism for disconnecting processors into separately acting instruction domains. This will now be complemented with a mechanism for connecting processors together to form a larger instruction domain. This mechanism for interconnecting processors can be used instead of the mechanism for segmenting the processor. Both mechanisms provide essentially the same capability—namely that of fitting the processor size to the size required by the program being run—but they have different properties and different costs. The former mechanism is suitable for running interdependent programs concurrently in instruction domains; the ability to merge or divide instruction domains in this mechanism appears to offer considerable programming flexibility. However, as the processor size increases, the number of instruction domains that request I/O increases, and the resulting "bottleneck" slows down the processor. A mechanism will be presented that circumvents this difficulty.

A connection network is desired to connect processors together. It must provide for correct interconnection of all rails and lines used in the processor. The maximum propagation delay time must be kept low, yet the interconnections possible in this network must be many. It would be further desirable that the network could be reduced to a tree structure, so that it could be physically realized in a structure with low transmission delay times.

The SW-structure is a connection network that is derived from a tree. In Figure 10, all of the tree above level 2 is reproduced and attached to the original tree at

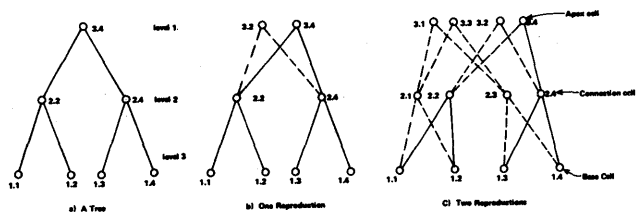


Figure 10—Formation of an SW-structure

level 2. In that figure, a tree structure can be restored by disconnecting links between levels one and two. In fact two tree structures can be obtained. For example, by cutting links between nodes (3, 2) and (2, 2) and again between nodes (3, 4) and (2, 4), one tree has nodes (3, 4), (2, 2), (1, 1) and (1, 2), and the other tree has nodes (3, 2), (2, 4), (1, 3) and (1, 4). In Figure 10c, all of the tree above level 3 has been reproduced again. Here, four tree structures can be obtained. In fact, for each partitioning of the nodes (1, 1), (1, 2), (1, 3) and (1, 4), a collection of disjoint trees can be found such that each tree has the nodes of each block of the partition. In essence, then, and SW-structure is a connection network that partitions a set of nodes, here at level 3, into blocks, and provides a tree structure for each block.

This process of reproducing can be done at each level of a tree having ℓ levels. It can be generalized to a tree whose fanout is f, just as easily as to the binary tree of Figure 10. Further, more than one reproduction, say $s - 1$ reproductions, can be made at each level. The resulting structure is a (uniform) *SW-structure* of ℓ levels, with fanout f and spread s. This structure will have $f^{\ell-1}$ *base nodes* (at the bottom of the structure) and $s^{\ell-1}$ *apex nodes* (at the top of the structure). By means of such an SW-structure, the set of $f^{\ell-1}$ base nodes can be partitioned into $s^{\ell-1}$ or fewer partition blocks. (We note that for $s = \ell = 2$, the structure is similar to a permutation switching network.[12])

The nodes of the SW-structure are cells. *Base cells* occupy base nodes, *apex cells* occupy apex nodes, and *connection cells* occupy the remaining nodes of the structure (Figure 10). The links are communication channels; each base or connection cell essentially connects at most one link towards the apex from it. Base and connection cells each have a *switching state* SS. $SS = 0$ if no link apex-ward link is connected, and $SS = i$ if the ith apex-ward link $i = 1, 2, \ldots S$, is connected. There connected links form one or more trees in the SW-structure.

The SW-structure is used to connect together processors and I/O channels, as in Figure 11, to form a processor system. Each base cell of the SW-structure connects to a root cell of a processor, or to a single cell which controls access to an I/O channel. A set of processors and I/O channels connected together in the SW-structure is a *processor block*. The entire processor block has the interconnection pattern of a tree. (See heavy line in Figure 11) In this tree, SW-structure cells appear to be permanently passive cells in their functional relation to cells in the processor; they have the channel and rail amplifiers that such a passive cell would have. In particular, the perimeter rails go around this tree, part of which is within the SW-structure, to provide a

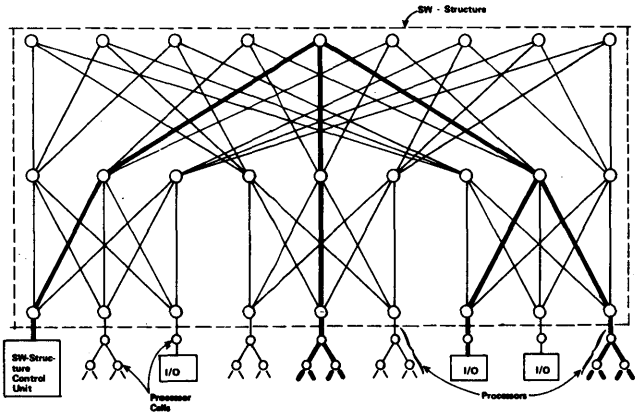


Figure 11—A processor system

linear ordering of cells in processors. Connection cells may have some hardware to detect faulty operation in them, since the SW-structure can operate in the presence of faulty cells, but this is not investigated here.

The SW-structure is manipulated by a control unit (Figure 11). Commands are sent to this unit as though it were an output channel. Exactly one instruction domain then, will be connected to this device, and it will send commands to it by sending out a string of data words while it is in the transfer enable mode as though the control unit were an I/O device. Meanwhile, other processor blocks operate without interruption.

We consider the connection and then the disconnection of a tree structure in the following discussion. Commands are sent to the control unit to connect a tree structure in the SW-structure. The three steps in connecting a tree are: (1) selecting some base nodes to which a suitable collection of processors and I/O units are attached, (2) choosing an apex node that can become a root cell of the tree structure which has all the selected base nodes in it, and finally, (3) arranging the switching state, SS, in base and connection cells to connect the tree structure.

The first step can be done in many ways. For example, each base node might have an address, and the control unit might use a small channel, to base cells only, in order to select base cells one at a time by addressing them. The second step must prevent an unwanted disconnection of trees which are being used to connect present processor blocks when the tree for this processor block is connected in step 3. For the second step, we provide a line in each link of the SW-structure. One at a time, each selected base cell broadcasts a "one" signal apex-ward on these lines. An SW-structure cell delimits this broadcast if it is already being used in a tree connecting together some processor block ($SS \neq 0$), and is therefore not available. All apex cells receiving "one" on this line have a chain of unused SW-structure cells to the selected base cell. When this procedure has been repeated for each base cell, of those apex cells that are connectable to each selected base cell, a hardware priority circuit among vertex cells selects one such cell to be the root of the tree.

For the third step of the problem, we use a second line in each link of the SW-structure. The selected apex cell broadcasts base-ward on this line, while on the line used in step 2, all selected root cells simultaneously broadcast apex-ward. Any SW-structure cell receiving a "one" from both an apex and base cell will set its switching state, SS, to connect that branch apex-ward from it on which the "one" signal from an apex cell arrived. If a tree structure exists to connect the processor block, this technique will connect it up.

Commands are sent to the control unit to disconnect a tree structure in a processor block which has completed its program. A third line is provided in each link of the SW-structure. By this line, any base cell can broadcast to exactly those cells used to connect the tree structure. To disconnect the tree, a command selects some base cell in the tree, and a second command causes this base cell to broadcast on this third line. All cells receiving a "one" signal on this line sets SS, the switch state, to 0, thereby disconnecting the tree. These cells now become available for connecting new tree structures.

We remark that a processor system has some useful properties of a computer network. Suppose that information for each area in the 2-D grid of Figure 12 were processed in a separate processor. By means of an SW-structure, the processor for area i, j could be connected to the processor for area $i + 1, j$ then $i - 1, j$ then $i, j + 1$ then $i, j - 1$. Meanwhile, the processor for area $i + 1, j$ could be connected to i, j then $i + 2, j$ then $i + 1, j + 1$, then $i + 1, j - 1$. Two dimensional

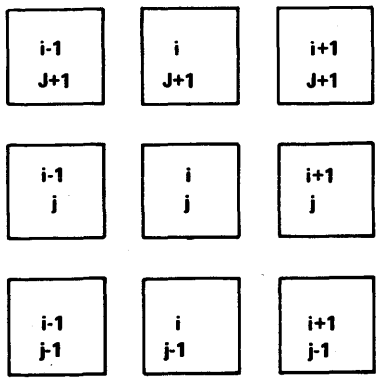


Figure 12—A 2D grid

differential equations, or picture processing, or other iterative parallel programs could be executed in parallel in this system. Further, higher degree grids can also be handled. This system appears to offer many advantages of a Holland space.[10]

CONCLUSIONS

A processor for Information Storage and Retrieval apparently must be large to be practical. We have presented two solutions to several problems associated with large processors. One solution is the segmentation of a processor, and the other is a system of processors. The architectures have low propagation delay, and provides for fast loading and unloading data in parallel from different I/O channels, and permit highly parallel programming. The diagnosis of cells was only lightly considered here. There are many indications that this diagnosis will be relatively simple. It is apparent that both architectures are capable of operating in the presence of some cells found to be faulty. Further, a graceful degradation of performance occurs when most cells are faulty. The results presented here apply to a number of associative memory processors, and may also apply to future iterative processors that broadcast instructions in a channel, even though they do not use the associative search operation as associative processors do.

ACKNOWLEDGMENTS

The author wishes to acknowledge the excellent criticism of his advisor, Professor R. P. Preparata, and the suggestions of Professor R. T. Chien, Professor S. Ray, and his fellow students at the Coordinated Science Laboratory, Urbana Illinois. The author further wishes to thank the University of Illinois fellowship committee and the Joint Services Electronics Program, contract DAAB-07-67-C-199, for their financial support. Finally the author wishes to thank Keith L. Doty, at the University of Florida, for reading the manuscript and offering many valuable suggestions.

REFERENCES

1 B A CRANE
Path finding with associative memory
IEEE Trans Computers Vol C-17 pp 691–693 July 1968

2 B A CRANE J A GITHENS
Bulk processing in distributed logic memory
IEEE Trans on Electronic Computers Vol EC-14 pp 186–196 April 1965

3 P M DAVIES
Design of an associative computer
Proc WJCC pp 109-117 March 1963

4 G ESTRIN R H FULLER
Algorithms for content-addressable memories
Ibid pp 118–130

5 J A FELDMAN P D ROVER
An algol-based associative language
Stanford Artifical Intelligence Project Memo AI-66 August 1 1968

6 C C FOSTER
Determination of priority structure in association memories
IEEE Trans Computers (Short Notes) Vol C-17 pp 788–789 August 1968

7 R H FULLER R M BIRD
An associative parallel processor with application to picture processing
1965 Fall Joint Computer Conference pp 105–116

8 R H FULLER G ESTRIN
Some applications for content addressable memories
Proc FJCC pp 495-505 November 1963

9 R S GAINS C Y LEE
An improved cell memory
IEEE Trans Electronic Computers Vol EC-14 pp 72-75 February 1965

10 J H HOLLAND
A universal computer capable of executing an arbitrary number of subprograms simultaneously
Proc FJCC pp 108–113 1959

11 K E IVERSON
A programming language
Wiley 1962

12 W H KANTZ K N LEVITT A WAKSMAN
Cellular interconnection arrays
IEEE Trans on Computers Vol C-17 pp 443-451 May 1968

13 C Y LEE
Intercommunicating cells, basis for a distributed logic computer
Proc of FJCC pp 130-136 1962

14 C Y LEE M C PAULL
A content addressable distributed logic memory with applications to information retrieval
Proc IEEE Vol 51 pp 924-932 June 1963

15 M H LEVIN
Retrieval of ordered lists from a content addressed memory
RCA Rev Vol 23 pp 215–229 June 1962

16 G J LIPOVSKI
The architecture of a large distributed logic associative processor
Coordinated Science Laboratory R-424 July 1969

17 B T McKEEVER
The associative memory structure
Proc FJCC Part I pp 371–388 1965

18 H O McMAHON A E SLADE
A cyrotron catalog memory system
Proc FJCC p 120 December 1956

19 R MORRIS
Scatter storage techniques
Communications of the ACM No II pp 38–44 January 1968

20 N K NATARAJAN P A V THOMAS
A multiaccess associative memory
IEE Trans on Computers Vol C-18 pp 424-428 May 1969

21 L G ROBERTS
Graphical communication and control languages
Second Congress on Information Systems Sciences Hot Spring Virginia 1964

22 D A SAVITT H H LOVE R E TROOP
ASP–A new concept in language and machine organization
Proc SJCC pp 87–102 1967

23 ———
Association-storing processor study
Hughes Aircraft Technical Report No TR-66-174 (AD-488538) June 1966

24 A E SLADE
The woven cryotron memory
Proc Int Symp on the Theory of Switching Harvard University Press 1959

25 J E SMATHERS
Distributed logic memory computer for process control
PhD Dissertation Oregon State University June 1969

26 J N STURMAN
An iteratively structured general-purpose digital computer
IEEE Trans on Computers Vol C-17 pp 2–9 January 1968

27 ———
Asynchronous operation of an iteratively structured general-purpose digital computer
Ibid pp 10–17

28 ———
An iteratively structured computer
PhD Dissertation Cornell University Itaca New York September 1966

29 P WESTON S M TAYLOR
Cylinders—A data structure concept based on rings
Coordinated Science Laboratory Report R-393 September 1968

30 S S YAU C C YANG
A nonbulk addition technique for associative processors
IEEE Trans on Electronic Computers Vol Ec-15 pp 938–941 December 1966

Application of invariant imbedding to the solution of partial differential equations by the continuous-space discrete-time method

by PAUL NELSON, JR.

*Oak Ridge National Laboratory**
Oak Ridge, Tennessee

INTRODUCTION

The continuous-space discrete-time (CSDT) method[1] of solving initial-boundary value problems for partial differential equations leads to two-point boundary-value problems for a system of ordinary differential equations. In order to solve such problems on an analog computer it is necessary to find an algorithm which expresses the desired solution in terms of initial-value problems. Various methods for accomplishing this are discussed in a recent survey article by Vichnevetsky.[2]

The invariant imbedding technique, which originally arose in connection with radiative transport problems, is essentially a method for converting a two-point boundary-value problem for a linear system of ordinary differential equations to an equivalent initial-value problem for an associated *nonlinear* system of ordinary differential equations. The purpose of this paper is to suggest the possibility of using invariant imbedding within the CSDT method, and to preliminarily explore some of the ramifications of this suggestion. Extensive references to work in invariant imbedding are given in the book by Wing,[3] and the articles by Bellman, Kalaba, and Wing,[4] by Bailey and Wing,[5] and by Nelson and Scott.[6]

For clarity and ease of exposition we shall attempt to illustrate the application of invariant imbedding to the CSDT method within the context of a simple example. The example selected is the standard one of time-dependent heat diffusion in one spatial dimension, but with rather special boundary conditions. The last section of the paper contains a discussion of possible extensions, as well as limitations, with pertinent references.

* Operated by Union Carbide Corporation for the U. S. Atomic Energy Commission

THE CSDT METHOD

We follow Vichnevetsky[7,8] in describing the application of the CSDT method to the time-dependent heat diffusion equation in one spatial dimension, x. The specific heat and conductivity are assumed independent of x, time, t, and temperature, u. The unit of time is selected so that the ratio of conductivity to specific heat is unity, and the unit of length is selected to have x varying between 0 and 1. The corresponding diffusion equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + S(x, t), \tag{1}$$

where the known function S determines the internal heat source. The temperature, $u(x, t)$, is required to satisfy the initial condition

$$u(x, 0) = u_0(x), \tag{2}$$

and the boundary conditions

$$u(0, t) \equiv 0, \tag{3}$$

$$u(1, t) = U(t), \tag{4}$$

where u_0 and U are given.

We introduce a positive discrete time step, Δt, and write $u_i(x)$ for $u(x, i\Delta t)$, $\bar{S}_i(x)$ for $S(x, i\Delta t)$. The fundamental idea of the CSDT method is to replace (1) by the system of equations

$$\frac{u_{i+1} - u_i}{\Delta t} = \theta\left[\frac{d^2u_{i+1}}{dx^2} + \bar{S}_{i+1}(x)\right] + (1 - \theta)\left[\frac{d^2u_i}{dx^2} + \bar{S}_i(x)\right], \qquad i = 0, 1, \cdots, \tag{5}$$

where θ is some constant. Equation (5) can be re-

written in the form

$$\frac{d^2u_{i+1}}{dx^2} - \frac{u_{i+1}}{\theta\Delta t} = -S_i(x), \qquad i = 0, 1, \cdots, \tag{6}$$

where

$$S_i(x) = \frac{u_i(x)}{\theta\Delta t} + \tilde{S}_{i+1}(x) + \frac{(1-\theta)}{\theta}\left[\frac{d^2u_i}{dx^2} + \tilde{S}_i(x)\right]. \tag{7}$$

Since u_0 is known from (2), we can find S_0 from (7), thence u_1 from (6) and the boundary conditions (3) and (4), which then determines S_1 from (7), etc. More practical methods of determining the S_i are discussed by Vichnevetsky.[7,8]

THE IMBEDDING FUNCTIONS

Inasmuch as the time index i plays no essential role for a while, we shall omit it until further notice. Thus we consider the second-order ordinary differential equation

$$\frac{d^2u}{dx^2} - \frac{u}{\theta\Delta t} = -S(x), \tag{8}$$

$S(x)$ given, and u to satisfy the boundary conditions

$$u(0) = 0, \tag{9}$$

$$u(1) = a, \tag{10}$$

where the constant a is given. This problem can, of course, be solved explicitly, up to an integral of $S(x)$, but our purpose here is to illustrate a technique rather than to solve a problem.

A well-known result, quite fundamental to our development, is that, for $\theta > 0$ and $y \neq 0$, the only solution of

$$\frac{d^2u}{dx^2} - \frac{u}{\theta\Delta t} = 0, \tag{11}$$

satisfying

$$u(0) = u'(y) = 0, \tag{12}$$

where the trivial solution $u(x) \equiv 0$. In order to use this result we henceforth assume $\theta > 0$.

Let $\mathfrak{U}$ be the class of nontrivial (i.e., not identically zero) functions satisfying (11) and (9). If $u, \bar{u} \in \mathfrak{U}$, then it follows from the fundamental result stated above that $u(x) = c\bar{u}(x)$ for some constant c. Consequently the functions

$$R(x) = u(x)/u'(x), \qquad u \in \mathfrak{U}, \tag{13}$$

$$T(x) = u'(0)/u'(x), \qquad u \in \mathfrak{U}, \tag{14}$$

are well defined; i.e., the values on the right-hand side are independent of the particular element $u \in \mathfrak{U}$.

In order to solve the problem (8)–(10), we need to introduce two additional auxiliary functions. Our introduction of these functions is motivated by the work of Wing,[9] who first rigorously applied invariant imbedding to inhomogeneous problems. We henceforth regard $S(x)$ as fixed, and defined for $0 \leq x \leq 1$. Consider that solution of (8) which satisfies the boundary conditions (12), where $y \in [0, 1]$ is arbitrary, and denote this function by $\tilde{u}(x, y)$ to indicate its dependence on the parameter y. Existence and uniqueness of $\tilde{u}(x, y)$, for arbitrary fixed y, is a standard result in the theory of ordinary differential equations.[10]

We denote the partial derivatives of functions of two variables by subscripts, the subscripts 1 and 2 indicating partial differentiation with respect to the first and second arguments, respectively. Let $E_r(x)$ and $E_l(x)$ be defined for $0 \leq x \leq 1$ by

$$E_r(x) = \tilde{u}(x, x), \tag{15}$$

$$E_l(x) = \tilde{u}_1(0, x). \tag{16}$$

The functions R, T, E_r, and E_l can be shown to satisfy the differential equations

$$R'(x) = 1 - \frac{R^2(x)}{\theta\Delta t}, \tag{17a}$$

$$T'(x) = -\frac{R(x)T(x)}{\theta\Delta t}, \tag{17b}$$

$$E_r'(x) = R(x)\left[-\frac{E_r(x)}{\theta\Delta t} + S(x)\right], \tag{17c}$$

$$E_l'(x) = T(x)\left[-\frac{E_r(x)}{\theta\Delta t} + S(x)\right], \tag{17d}$$

and to have the initial values

$$T(0) = 1, \tag{18a}$$

$$R(0) = E_r(0) = E_l(0) = 0. \tag{18b}$$

The initial values (18) are easy consequences of the above definitions, as are the differential equations (17a–b) for R and T. The derivation of (17c–d) is somewhat lengthy and difficult to motivate, although basically quite simple. It is outlined in the appendix, in order to avoid having at this point a lengthy digression from our main purpose, namely solution of (8)–(10).

SOLUTION OF EQUATIONS (8)–(10)

We now suppose the functions R, T, E_r, and E_l are known, and attempt to construct the solution of the original problem (8)–(10) from these functions. Let $u(x)$ be the solution of (8)–(10), suppose $\tilde{u}(x, y)$ is as above, with y anywhere in $[0, 1]$, and define $\varphi(x, y)$ by

$$\varphi(x, y) = u(x) - \tilde{u}(x, y). \tag{19}$$

Then φ, considered as a function of x for fixed y, is either identically zero, or is in the class $\mathfrak{U}$, defined above. In either event the identities

$$\varphi(x, x) = \varphi_1(x, x)R(x), \tag{20}$$

and

$$\varphi_1(0, x) = \varphi_1(x, x)T(x) \tag{21}$$

follow from the definitions (13) and (14) and the fundamental result stated at the beginning of the preceding section.

From (19), (20), (19) (again), the definition of $\tilde{u}$, and (15), we obtain the identity

$$\begin{aligned} u(x) &= \varphi(x, x) + \tilde{u}(x, x) \\ &= \varphi_1(x, x)R(x) + \tilde{u}(x, x) \\ &= [u'(x) - \tilde{u}_1(x, x)]R(x) + \tilde{u}(x, x) \\ &= u'(x)R(x) + E_r(x). \end{aligned} \tag{22}$$

Similarly the identity

$$u'(0) = u'(x)T(x) + E_l(x) \tag{23}$$

is easily established.

The identities (22) and (23) are the key results which enable us to solve the original problem. First note that taking $x = 1$ in (22) and taking account of (10) enables us to express $u'(1)$ in terms of known quantities, to wit

$$u'(1) = [a - E_r(1)]/R(1). \tag{24}$$

With this result in hand, one can solve the original problem as an initial value problem, the initial data being given at $x = 1$ by (10) and (24). This is the algorithm suggested by Wing,[3] and used extensively by Rybicki and Usher.[11] If the original problem (8) were computationally stable in the backward direction, then this would probably be the best procedure; however one of the solutions of the homogeneous equation associated with (8) is $\exp(-x/\sqrt{\theta\Delta t})$, which shows that (8) is very unstable in the backward direction for the small values of Δt which are needed to keep the time discretization error reasonably small. There is a second way to find $u(x)$, due essentially to Scott[12] (see also Nelson and Scott[6]) which avoids this difficulty, and which we now describe.

Letting $x = 1$ in (23), we find the equality

$$u'(0) = u'(1)T(1) + E_l(1), \tag{25}$$

which expresses $u'(0)$ in terms of known quantities, $u'(1)$ being known from (24). Equation (23) then gives $u'(x)$ in terms of known quantities, and (22) yields an expression for $u(x)$ in terms of knowns. The final expression is

$$\begin{aligned} u(x) = a\frac{R(x)}{R(1)}\frac{T(1)}{T(x)} - E_r(1)\frac{R(x)}{R(1)}\frac{T(1)}{T(x)} \\ + \frac{[E_l(1) - E_l(x)]}{T(x)}R(x) + E_r(x). \end{aligned} \tag{26}$$

The reason for this peculiar grouping of terms will become apparent in the next section.

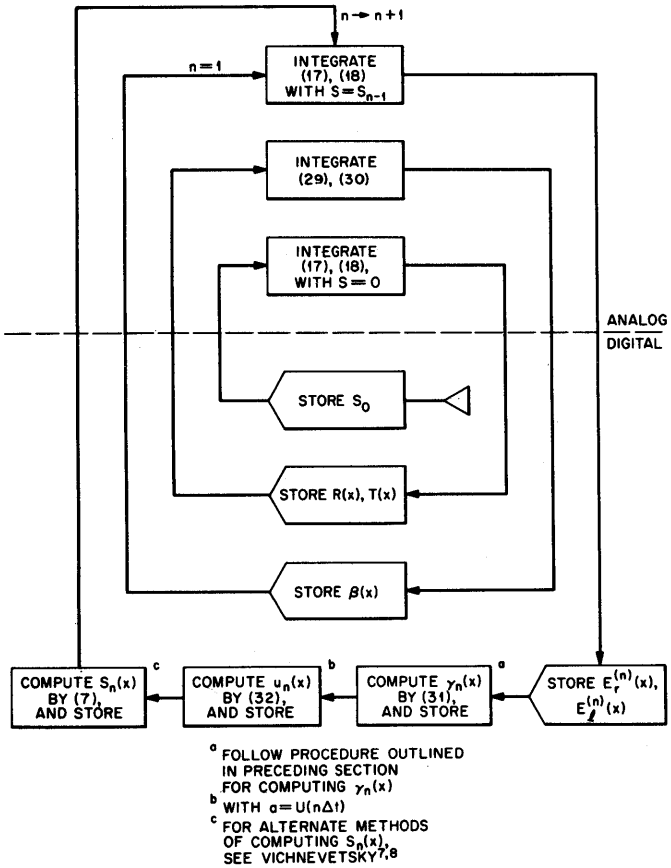
COMPUTATIONAL CONSIDERATIONS

On a digital computer it would be quite feasible to obtain u from (26), with R, T, E_r, and E_l obtained by integrating (17) subject to (18). In fact similar approaches have been implemented digitally, and have been shown to be quite stable computationally.[13] However this formula is not appropriate for analog solution of (17)–(18), at least in the context of the CSDT method. The reason is that (26) contains ratios which must be known accurately to yield an accurate value of $u(x)$, while the quantities entering these ratios are too small to be determined accurately from analog solution of (17)–(18). In order to see this let us look somewhat more closely at (17)–(18).

The function $R(x)$, which is determined by (17a) and (18b), will be an increasing function, with slope approximately unity near $x = 0$, and the slope decreasing as x increases, with $R(x)$ asymptotically approaching $\sqrt{\theta\Delta t}$ as $x \to \infty$. In fact $R(x)$ should be, for practical purposes, approximately equal to $\sqrt{\theta\Delta t}$ for x greater than a few multiples of $\sqrt{\theta\Delta t}$. (This follows from the inequalities

$$\begin{aligned} \sqrt{\theta\Delta t}\exp(-2x/\sqrt{\theta\Delta t}) \leq \sqrt{\theta\Delta t} - R(x) \\ \leq \sqrt{\theta\Delta t}\exp(-x/\sqrt{\theta\Delta t}), \end{aligned}$$

which can be established fairly easily.) Now θ is of the order of unity, and Δt must be taken fairly small, say certainly $\Delta t \lesssim .01$, in order to keep the time discretization error reasonably small. Thus we conclude that $R(x)$ will rise quite rapidly from 0 at $x = 0$ to (almost) $\sqrt{\theta\Delta t}$ as x increases, and that $R(x) \approx \sqrt{\theta\Delta t}$ for $\epsilon \leq x \leq 1$, where ϵ is small.



Block Diagram for the Application of Invariant Imbedding to the Hybrid Solution of Equations (1)-(4) by the CSDT Method.

Figure 1—Block diagram for the application of invariant imbedding to the hybrid solution of Equations (1)–(4) by the CSDT method

Considering now (17b) and (18a) we find, since $R(x) \approx \sqrt{\theta\Delta t}$ for most values of x, that

$$T(x) \approx \exp(-x/\sqrt{\theta\Delta t}), \tag{27}$$

the approximation being quite good for x greater than a few factors of $\sqrt{\theta\Delta t}$, and certainly for x near 1. Now (27) shows that the largest values of the ratio $T(1)/T(x)$, which appears in the first two terms of (26), occur near $x = 1$. But (27) also shows that both $T(1)$ and $T(x)$, x near 1, are down at least 3–4 orders of magnitude from the maximum value $T(0) = 1$. Consequently, because of the low intensity resolution of analog computers, these quantities will both essentially be reported as zero by analog solution of (17)–(18). Thus the most inaccurate values of $T(1)/T(x)$ are obtained for exactly those x at which accuracy is most important.

In order to get around the above difficulty we introduce the function $\beta(x)$, defined by

$$\beta(x) = T(1)/T(1-x), \qquad 0 \le x \le 1. \tag{28}$$

Then β is determined by the initial value problem

$$\beta'(x) = -R(1-x)\beta(x)/\theta\Delta t, \tag{29}$$

$$\beta(0) = 1. \tag{30}$$

In order to solve (29)–(30), $R(x)$ must be available for $0 \le x \le 1$ from previous solution of (17a) subject to $R(x) = 0$. Analog computer integration of (29)–(30) will produce reliable values of $\beta(1-x)$ for x near 1, which is precisely where this factor is most important in (26).

At first look it might be thought that similar difficulties would be associated with the factor

$$\gamma(x) = [E_l(1) - E_l(x)]/T(x) \tag{31}$$

in (26). However a little further study shows that $\gamma(x)$ is actually a decreasing function of x, and that $E_l(x)$ approaches $E_l(1)$ somewhat faster than $T(x)$ approaches zero. Consequently $\gamma(x)$ will be zero, within analog resolution, before $T(x)$, and the proper computational procedure is to set $\gamma(x) = 0$ for larger x by internal logic.

In terms of β and γ, (26) becomes

$$u(x) = a\frac{R(x)}{R(1)}\beta(1-x) - E_r(1)\frac{R(x)}{R(1)}\beta(1-x) + \gamma(x)R(x) + E_r(x). \tag{32}$$

We note in passing the interesting fact that the first three terms in (32) are important only in relatively thin boundary layers, near $x = 1$, $x = 1$, and $x = 0$, respectively. The first term represents the effect of the imposed boundary condition at $x = 1$, and the remaining terms stem from the source function. The behavior near $x = 0$ is dominated by the third term, and near $x = 1$ by the first term, except when $a = 0$ the combined second and fourth terms dominate near $x = 1$, in spite of the fact that they cancel exactly at $x = 1$. The behavior far (i.e., a few multiples of $\sqrt{\theta\Delta t}$) from either boundary is dominated by the fourth term, $E_r(x)$.

HYBRID IMPLEMENTATION

Figure 1 shows a block diagram of one possible hybrid implementation of the method presented here. We have written $E_r^{(n)}$ and $E_l^{(n)}$ for E_r and E_l corresponding to $S = S_n$, and γ_n for γ corresponding to $E_l = E_l^{(n)}$.

Note that R, T, and β are retained permanently in

digital storage, but that they are generated anew in the analog section each time (17c–d) are to be integrated to obtain $E_r^{(n)}$ and $E_l^{(n)}$. This procedure is intended to minimize the time consumed by D/A data transmission, and to make available accurate information regarding the high frequency components of R and T during the integration of (17c–d). The latter consideration is particularly important near $x = 0$, where all of the imbedding functions are changing quite rapidly. It is true that the low frequency pass band of the digital section does not permit accurate knowledge of the high frequency components of S_n in integrating (17c–d), but these are probably relatively less important in most cases than are the high frequency components of R and T.

As a programming note we remark that the procedure indicated in Figure 1 never requires E_l and γ to be available at the same time, and therefore these two variables can occupy the same locations in digital storage. The same comment holds for the variables u and E_r.

EXTENSIONS AND LIMITATIONS

The invariant imbedding technique may be thought of, at least with regard to its application in the CSDT method, as fundamentally applying to problems of the form

$$u'(x) = A(x)u(x) + B(x)v(x) + S_1(x), \quad (33a)$$

$$v'(x) = C(x)u(x) + D(x)v(x) + S_2(x), \quad (33b)$$

with two-point boundary conditions of the type

$$u(0) = \alpha v(0) + \beta, \quad (34a)$$

$$v(x_0) = \gamma u(x_0) + \delta. \quad (34b)$$

Here x is to range between 0 and x_0, u and v are vectors of finite (but not necessarily equal) length, A, B, etc., and α, β, etc. are respectively matrix functions and constant matrices of the appropriate sizes. The problem (8)–(10) is of this form after the substitution $v = u'$. Details of the imbedding functions and their application in solving (33)–(34) are given by Scott[12] and by Nelson and Scott.[6]

The statements of the preceding paragraph imply that the invariant imbedding solution of the CSDT equations can be applied, at least in principle, to any linear partial differential equation, provided the associated boundary conditions in the continuous variable can be put in the linear inhomogeneous form (34). The method does not apply directly to problems in which either the differential equation or the boundary conditions are nonlinear. However the other methods[2] proposed for solving the ordinary differential equations of the CSDT method share this defect, except for the shooting method, and the latter is well known to have serious stability defects. Vichnevetsky[14] has suggested that nonlinear problems be solved by an iterative predictor–corrector technique, with the decomposition method to be used to solve an approximating problem linearized about the predicted solution. In a similar vein, but with digital application in mind, Allen, Wing, and Scott[15] have considered the idea of solving nonlinear problems by the application of invariant imbedding to an appropriate sequence of linearized problems.

In conclusion, we believe that the method presented here shows sufficient promise to warrant further investigation. Such investigations should include a quantitative comparison of the present method with other commonly used techniques for solving the CSDT equations, with due regard to both effectiveness and computer requirements for frequently occurring types of problems. We intend to pursue such a study, and hope to communicate the results elsewhere.

REFERENCES

1 S H JURY
Solving partial differential equations
Industrial and Engineering Chemistry 53 p 177–180 1961

2 R VICHNEVETSKY
Analog hybrid solution of partial differential equations in the nuclear industry
Simulation 11 p 269–281 1968

3 G M WING
An introduction to transport theory
John Wiley and Sons Inc New York 1962

4 R BELLMAN R KALABA G M WING
Invariant imbedding and mathematical physics I—Particle processes
Journal Math Phys 1 p 280–308 1960

5 P B BAILEY G M WING
Some recent developments in invariant imbedding with applications
Journal Math Phys 6 p 453–462 1965

6 P NELSON JR M R SCOTT
Internal values in particle transport by the method of invariant imbedding
SC-RR-69-344 Sandia Corporation Albuquerque New Mexico 1969
Submitted to Journal Math Phys

7 R VICHNEVETSKY
A new stable computing method for the serial hybrid computer integration of partial differential equations
Proceedings SJCC Atlantic City New Jersey Thompson Book Company 1968

8 R VICHNEVETSKY
Application of hybrid computers to the integration of partial differential equations of the first and second order
IFIP Edinburgh A68-A75 1968

9 G M WING
Invariant imbedding and transport problems with internal sources
Journal Math Anal Appl 13 p 361–369 1966
10 E A CODDINGTON N LEVINSON
Theory of ordinary differential equations
McGraw–Hill Company esp chapter 7 1955
11 G B RYBICKI P D USHER
The generalized Riccati transformation as a simple alternative to invariant imbedding
Ap J 146 p 871–879 1966
12 M R SCOTT
Invariant imbedding and the calculation of internal values
J Math Anal Appl 28 p 112–119 1969
13 M R SCOTT
Numerical solution of unstable initial value problems by invariant imbedding
SC-RR-69-343 Sandia Laboratories Albuquerque New Mexico 1969
14 R VICHNEVETSKY
Serial solution of parabolic partial differential equations: The decomposition method for non-linear and space-dependent problems
Simulation 13 p 47–48 1969
15 R C ALLEN JR G M WING M R SCOTT
Solution of a certain class of nonlinear two-point boundary value problems
J Comp Phys 4 p 250-257 1969

APPENDIX

Recall that $\tilde{u}(x, y)$ is defined to satisfy the differential equation (8) and the boundary conditions (12), as a function of x. These defining conditions can be written respectively as

$$\tilde{u}_{11}(x, y) - \frac{\tilde{u}(x, y)}{\theta \Delta t} = -S(x), \qquad \text{(A-1)}$$

and the identities

$$\tilde{u}(0, y) = \tilde{u}_1(y, y) = 0. \qquad \text{(A-2)}$$

To obtain a differential equation for E_r, first note that (15) and (A-2) yield

$$E_r'(x) = \tilde{u}_1(x, x) + \tilde{u}_2(x, x) = \tilde{u}_2(x, x). \qquad \text{(A-3)}$$

If we could express $\tilde{u}_2(x, x)$ in terms of E_r and known functions of x, then (A-3) would give a differential equation for E_r. This is the objective in the next paragraph.

If (A-1) is differentiated with respect to y, the result can be put in the form

$$\tilde{u}_{211}(x, y) - \frac{\tilde{u}_2(x, y)}{\theta \Delta t} = 0,$$

after using equality of mixed partials. Equation (A-2) gives

$$\tilde{u}_2(0, y) = 0.$$

The last two equations imply that, for any $y \in [0, 1]$, either $\tilde{u}_2(x, y)$ as a function of x is in the class $\mathfrak{U}$ defined above, or $\tilde{u}_2(x, y)$ is identically zero. In either case we have the identity

$$\tilde{u}_2(x, y) = \tilde{u}_{21}(x, y) \frac{u(x)}{u'(x)}, \qquad u \in \mathfrak{U},$$

in $x, y \in [0, 1]$. If we set $y = x$ in this identity, and recall (13), then we find

$$\tilde{u}_2(x, x) = \tilde{u}_{21}(x, x) R(x).$$

But differentiation of the identity $\tilde{u}_1(x, x) = 0$ in (A-2), and application of (A-1) and (15) gives

$$\begin{aligned} \tilde{u}_{21}(x, x) &= -\tilde{u}_{11}(x, x) \\ &= \frac{-\tilde{u}(x, x)}{\theta \Delta t} + S(x) \\ &= \frac{-E_r(x)}{\theta \Delta t} + S(x). \end{aligned}$$

The two equations immediately preceding give an expression for $\tilde{u}_2(x, x)$ in terms of R, E_r, and S. If this expression is substituted into (A-3), the result is the differential equation (17c) for E_r. A similar development shows that

$$\begin{aligned} E_l'(x) &= \tilde{u}_{12}(0, x) \\ &= \tilde{u}_{21}(x, x) \frac{u'(0)}{u'(x)} \\ &= -\tilde{u}_{11}(x, x) \frac{u'(0)}{u'(x)} \\ &= \left[S(x) - \frac{\tilde{u}(x, x)}{\theta \Delta t} \right] \frac{u'(0)}{u'(x)} \\ &= T(x) \left[\frac{-E_r(x)}{\theta \Delta t} + S(x) \right], \end{aligned}$$

where u is an arbitrary element of $\mathfrak{U}$. This gives the equation (17d) for E_l.

An initial value formulation of the CSDT method of solving partial differential equations

by VENKATESWARARAO VEMURI

Purdue University
Lafayette, Indiana

INTRODUCTION

Numerical methods of solving partial differential equations (PDEs) using analog or hybrid computers fall into three broad categories. Assuming, for concreteness, that one of the independent variables is time and the rest are spatial, the continuous-space and discrete-time (or CSDT) methods envisage to keep the space-like variable continuous and discretize the time-like variable. Similarly, the terms discrete-space and continuous time (DSCT) and discrete-space and discrete-time (DSDT) approximations are self-explanatory. For a one-space dimensional PDE, for instance, both the CSDT and DSCT approximations yield a set of ordinary differential equations while the DSDT approximations lead to a set of algebraic equations. Because of the inherent need to handle a continuous variable, both CSDT and DSCT approximations lend themselves well for computation on analog or hybrid computers. Indeed, several analog and hybrid computer implementations of all these three methods are currently in vogue each method claiming to be superior in some respect to the others. However, it was the CSDT method that showed great promise and produced little results. The purpose of this paper is to present another alternative to this problem.

One of the fundamental advantages of the CSDT method over others is its ability to handle moving boundaries. This can be readily achieved by controlling the analog computer's integration interval since the problem space variable is represented by computer-time. A second advantage is that the analog hardware requirements of the CSDT method are very modest because a relatively small analog circuit is time-shared to solve the entire problem. With the advent of modern high-speed iterative analog and hybrid computers the above promises of the CSDT method appeared to be almost within the reach.[1,2]

In practice, however, considerable difficulties were encountered in obtaining dependable results using the CSDT method.[2,3] The major difficulty is that the CSDT methods are inherently unstable. Methods that were proposed to circumvent this stability problem are either conceptually wrong or impose additional computational burdens making their efficiency debatable. A second difficulty with the CSDT methods is that the basic spatial sweep from boundary to boundary, at each discrete time level, yields a two point boundary value problem (TPBVP) which in turn has to be solved iteratively. It is not clear, at the outset, whether any advantage gained by time-sharing of the analog hardware is really tangible when compared to the price paid in solving a TPBVP. A third disadvantage is that the CSDT method is essentially limited to handle problems in one space dimension only.

This paper suggests a new alternative which still adopts the basic CSDT procedure but results in an initial-value problem. By this formulation the first two difficulties cited in the preceding paragraph are eliminated. This paper still treats a one-space-dimensional problem and no attempt was made here to extend the concept to higher dimensions. However, it is not quite inconceivable to extend this technique to higher dimensions by using this in conjunction with an alternating direction iterative method.

STATEMENT OF THE PROBLEM

Consider the simple diffusion equation

$$\frac{\partial^2 U}{\partial x^2} = \frac{\partial U}{\partial t}\,; \quad U = U(x, t) \tag{1}$$

with the initial conditions

$$U(x, 0) = U_0(x) = f(x)\,; \quad 0 \leq x \leq 1 \tag{2}$$

and without loss of generality[5,7] with the homogeneous boundary conditions

$$\left.\begin{aligned} U(0,t) &= 0 \\ U(1,t) &= 0 \end{aligned}\right\} \tag{3}$$

A CSDT approximation to (1), (2) and (3) can be written, as usual, by using a backward difference approximation for the time-derivative. Specifically, at time $t = t_k$, using a simple difference scheme, Eq. (1) can be approximated by

$$LU_k(x) = \frac{1}{(\Delta t)}[U_k(x) - U_{k-1}(x)]; \quad 0 \leq x \leq 1 \tag{4}$$

where $L \triangleq d^2/dx^2$ is a differential operator and (Δt) is the size of the time step taken. With this approximation, the auxilary conditions (2) and (3) take the form

$$U_0(x) = f(x) \tag{5}$$

$$\left.\begin{aligned} U_k(0) &= 0 \\ U_k(1) &= 0 \end{aligned}\right\} \tag{6}$$

The classical method of implementing the CSDT method is to solve (4) on an analog computer with the initial condition (5) and the boundary conditions (6). However, equations (4), (5) and (6) constitute a TPBVP as such $U_k(x)$ for $0 \leq x \leq 1$ at any time level $t = t_k$ cannot be obtained in a single computer run; an iterative procedure is required to determine $U_k(x)$ at each $t = t_k$. This iterative procedure is often performed using either a trial and error procedure or by using a search technique such as the steepest descent method. Under such circumstances, scaling limitations of analog computers place severe restrictions on the region of search making them unattractive. Coupled with the inherent instability of the analog computer circuit solving (4), this necessity to solve a TPBVP at each time level is therefore the major drawback of the conventional CSDT method.

FORMULATION OF INTEGRAL EQUATION

The initial value formulation starts once again with equations (4), (5) and (6). Instead of solving them as a TPBVP, equations (4) through (6) are first transformed into an equivalent integral equation of the Fredholm type. The first step of this procedure, which can be found in any standard work,[4-7] is to determine a Green's function of the differential operator L in (4) that also satisfies the homogeneous boundary conditions in (6). Specifically, a Green's function for the operator $L = d^2/dx^2$ that satisfies the homogeneous boundary conditions in (6) can be written as

$$K(x,y) = \begin{cases} x(1-\eta); & 0 \leq x \leq \eta \leq 1 \\ \eta(1-x); & 0 \leq \eta \leq x \leq 1. \end{cases} \tag{7}$$

It is important to note that the Green's function has one form for $x < \eta$ and another for $\eta < x$ and that in each semi-interval it has a structure of the product of a function of x alone and a function of η alone. Such a structure is called semi-degenerate, which can greatly simplify the problem. If the Green's function $K(x, \eta)$ obtained is not degenerate or semi-degenerate, it can always be approximated, to any desired degree of accuracy, by a semi-degenerate kernel using standard techniques.[5-7] Therefore, the procedure presented here is not good for any nondegenerate kernel.

Solution of the TPBVP described by (4), (5) and (6) can now be written in terms of the Green's function (7) as

$$U_k(x) = \frac{1}{(\Delta t)}\int_0^1 K(x,\xi)U_{k-1}(\xi)\,d\xi - \frac{1}{(\Delta t)}\int_0^1 K(x,\xi)U_k(\xi)\,d\xi \tag{8}$$

or

$$U_k(x) = f_{k-1}(x) + \lambda\int_0^c K(x,\xi)U_k(\xi)\,d\xi \tag{9}$$

where $f_{k-1}(x)$ is the first term on the right hand side of (8) and can be explicitly evaluated because $U_{k-1}(x)$ represents a solution obtained at the preceding time level $t = t_{k-1}$. The terms λ and c in (9) are defined by

$$\lambda = -\frac{1}{(\Delta t)}; \quad c = 1$$

and are introduced merely for convenience and generality.

Equation (9) is a Fredholm integral equation of the second kind in its most familiar format. In (9), $f_{k-1}(x)$ is called the *free term*, λ the *parameter* and $U(x, \xi)$ is called the *kernel*. Without going into the details of a proof, let it be stated that for a well-posed problem, the solution $U(x, t)$ of the given PDE can be approximated by the sequence of functions $U_k(x)$ which are indeed the solution of the above integral equation.

This procedure of transforming the given PDE into an equivalent Fredholm type integral equation was apparently suggested also by Chan[8] in a recent paper but he adopts an *iterative* procedure to solve the integral equation.

SOLUTION OF THE INTEGRAL EQUATION

The next computational step is to solve the integral equation presented in (9) for $U_k(x)$. Classical methods of solving (9) are essentially iterative in nature[9] and so are not suitable for real-time operation. Furthermore, analog computers are ideally suited for solving problems with prescribed initial conditions. It, therefore, is logical to search for methods of transforming integral equations into sets of ordinary differential equations with prescribed initial conditions. Such a method was recently suggested by Kalaba.[10]

Kalaba's method is essentially one of treating the interval of integration $(0, c)$ as a variable rather than as a constant. By regarding the solution at a fixed point as a function of the interval of integration (now being treated as a variable), a set of ordinary differential equations with a complete set of initial conditions can be obtained. With a knowledge of the solution for one interval length, it is now easy to generate solutions for other interval lengths or for any interval length using this equation as a vehicle. Furthermore, the set of ordinary differential equations with prescribed initial conditions can be solved very easily on an analog computer.

Equation (9) is the starting point for the formulation of the initial-value problem. Treating the interval $(0, c)$ as a variable, (9) can be rewritten as

$$U_k(x, \tau) = f_{k-1}(x) + \int_0^\tau K(x, \xi) U_k(\xi, \tau)\, d\xi; \qquad 0 \leq x \leq \tau \quad (10)$$

It is assumed that (10) has a solution for $\tau \leq c$. For τ sufficiently small and

$$0 \leq x \leq \tau \quad (11)$$

the solution $U_k(x, \tau)$ of (10) can be proved (see appendix) to be identical to the solution of the set of equations *defined* by (12) through (20).

$$G(\tau) \triangleq (1 - \tau) + \tau r(\tau) \quad (12)$$

$$\frac{dr(\tau)}{d\tau} \triangleq [G(\tau)]^2 \quad (13)$$

$$\frac{de(\tau)}{d\tau} \triangleq G(\tau)[f_{k-1}(\tau) + (1 - \tau)e(\tau)]; \qquad \tau > 0 \quad (14)$$

with the initial conditions at $\tau = 0$ given by

$$r(\tau = 0) = r(0) = 0 \quad (15)$$

$$e(\tau = 0) = e(0) = 0 \quad (16)$$

and

$$\frac{dJ(x, \tau)}{d\tau} = G(\tau)[\tau \cdot J(x, \tau)]; \quad \tau > x \quad (17)$$

$$\frac{dU_k(x, \tau)}{d\tau} = [f_{k-1}(\tau) + \tau e(\tau)]J(x, \tau) \cdot \tau; \quad \tau > x \quad (18)$$

with the initial condition at $\tau = x$ given by

$$J(x, \tau = x) = (1 - x) + xr(x) \quad (19)$$

and

$$U_k(x, \tau = x) = f_{k-1}(x) = xe(x) \quad (20)$$

COMPUTATIONAL PROCEDURE

Equations (12) through (20) can now be solved using an analog computer or a hybrid computer. The various computational stages are indicated below.

Step 1. Solve (13) and (14) on an analog computer over the interval $0 \leq \tau \leq x$ by treating τ as computer time. Initial conditions for this computer run are given by (15) and (16) respectively.

Step 2. After integrating until time $\tau = x$, the analog computer is placed in HOLD mode and the solutions r and e, at $\tau = x$, obtained in step 1 are used to evaluate the expressions in (19) and (20). These values will be useful as initial conditions while solving (17) and (18) in the next step.

Step 3. At time $\tau = x$, and after (19) and (20) are evaluated equations (17) and (18) are adjoined to the original set (13) and (14) and both sets are integrated over the interval $x \leq \tau \leq c$ by putting the analog computer back in COMPUTE mode. During this phase of integration, the *initial conditions* of the additional set are the values of J and U_k evaluated *not* at $\tau = 0$ but at $\tau = x$. This is precisely the reason and purpose of the computation in step 2.

Step 4. The output of the integrator solving equation (18) in $U_k(x, \tau)$ and this is the solution of (9) at the argument x. This is also the solution of the PDE (1) at a particular time level $t = t_k$.

DISCUSSION

Initial-value problems are conceptually simple, computationally easy to solve and are susceptible for simulation studies. Simulation inherently involves trial and error experimentation in which the validity of a model is verified; sensitivity to environment is explored and variation of performance due to parameter changes evaluated. Such problems come under the classical heading of inverse problems—that is, problems where a system's performance is known from a measured set

of observations and the nature of the system is to be determined. While solving such inverse problems by using such search techniques as gradient methods, it is often necessary to solve not only the dynamic equation of the system, such as (1), but also an additional equation called the derived equation. This is not a mere doubling of computational effort as it appears at first sight. The computational effort required in the evaluation of the gradient increases very fast if the derived equation is an adjoint equation posed as a final value problem. It is precisely in bottleneck situations like this that an initial-value formulation comes in handy.

A second possible application of this method would be in on-line control or identification of distributed parameter systems.

Implementation of this method, particularly when the kernel has no simple structure requires some degree of sophistication in the analog system. If the Green's function (or Kernel) contains, or is approximated by, expressions that are sums of products of a large number of terms then the analog circuit generally contains a large number of multipliers. This may make the scaling a little more difficult. Finally, computation from step 2 to step 3 requires a degree of sophistication in the analog switching system. Many present generation hybrid computer systems can indeed handle most of these requirements.

No attempt was made in this paper to present a procedure that can be applied to any partial differential equation. Similarly no assumptions were made that would restrict the procedure to the simple case presented. In the general case, an easy procedure is required to obtain equations (12) through (20) from (10). Material filling these gaps and results supporting this procedure will be presented in a subsequent paper.

ACKNOWLEDGMENTS

The author wishes to express his thanks to Professor R. Kalaba who first introduced the procedure presented in the appendix, to solve an integral equation. He also wishes to express his gratitude to Professor George A. Bekey of the University of Southern California who provided support during the initial stages of this work. provided support during the initial stages of this work from AFOSR–1018–67C.

REFERENCES

1 H S WITSENHAUSEN
Hybrid solution of initial value problems for partial differential equations
MIT Electronics Systems Lab DSR 9128 Memo No 8 August 1964

2 R VICHNEVETSKY
A new stable computing method for the serial hybrid computer integration of partial differential equations
Proc Spring Joint Computer Conference Vol 33 Pt 1 pp 565–574 1968

3 H H HARA W J KARPLUS
Application of functional optimization techniques for the serial hybrid computer solution of partial differential equations
Proc Fall Joint Computer Conference Vol 33 pt 1 pp 565–574 1968

4 R E BELLMAN
Introduction to the mathematical theory of control processes
Academic Press 1967 Vol 1

5 F B HILDEBRAND
Methods of applied mathematics
Prentice Hall 1952

6 H H KAGIWADA R E KALABA
A practical method for determining Green's function using Hadamard's variational formula
J Optimization Theory Appl Vol 1

7 F G TRICOMI
Integral equations
Interscience 1963

8 S-K CHAN
The serial solution of the diffusion equation using nonstandard hybrid techniques
IEEE Transactions on Computers Vol 1-8 No 9 pp 786–799 September 1969

9 G A BEKEY W J KARPLUS
Hybrid computation
Wiley 1968

10 H H KAGIWADA R E KALABA
An initial value theory for Fredholm integral equations with semi-degenerate kernels
Rand Corporation Memorandum RM-5602-PR April 1968

APPENDIX

Outline of the initial-value formulation

Step 1: The proof starts with a realization that if $\Phi(x, \tau)$ is a solution of the integral equation

$$\Phi(x, \tau) = K(x, \tau) + \int_0^\tau K(x, \xi)\Phi(\xi, \tau)\, d\xi, \qquad 0 \leq x \leq \tau \quad \text{(A1)}$$

then

$$W(x, \tau) \triangleq \Phi(x, \tau)U(\tau, \tau) \quad \text{(A2)}$$

is a solution of the equation defined by

$$W(x, \tau) \triangleq K(x, \tau)u(\tau, \tau) + \int_0^\tau K(\lambda, \xi)W(\xi, \tau)\, d\xi \quad \text{(A3)}$$

A proof of this statement can be found in any standard book on integral equations.[5,7]

Step 2: To prove that the integral equation (10) is equivalent to the set of differential equations (12)

through (20), equation (10) is first differentiated with respect to τ. Denoting derivatives with respect to τ by primes, this differentiation yields

$$U_k'(x, \tau) = K(x, \tau) U_k(\tau, \tau) + \int_0^\tau K(x, \xi) U_k'(\xi, \tau)\, d\xi \tag{A4}$$

If $U_k'(x, \tau)$ is identified with $W(x, \tau)$, equation (A4) is identical to (A3). Therefore, the solution $U_k'(x, \tau)$ of Eq. (A4) can be written as

$$U_k'(x, \tau) = \Phi(x, \tau) U(\tau, \tau); \quad 0 \leq x \leq \tau \tag{A5}$$

where $\Phi(x, \tau)$ is the solution of (A1).

Step 3: Directing attention once again on (A1) and replacing the kernel $K(x, \tau)$ by its semi-degenerate approximation, namely

$$K(x, \tau) = \begin{cases} x(1 - \tau); & 0 \leq x \leq \tau \\ (1 - x)\tau; & 0 \leq \tau \leq x \end{cases} \tag{A6}$$

equation (A1) can be written as

$$\Phi(x, \tau) = \tau(1 - x) + \int_0^\tau K(x, \xi) \Phi(\xi, \tau)\, d\xi; \quad x \leq \tau \tag{A7}$$

$$= \tau J(x, \tau); \tag{A8}$$

where $J(x, \tau)$ is defined by the integral equation

$$J(x, \tau) \triangleq (1 - x) + \int_0^\tau K(x, \xi) \Phi(\xi, \tau)\, d\xi \tag{A9}$$

Step 4: If $J(x, \tau)$ can be determined, then using (A8) the function $\Phi(x, \tau)$ can be obtained which in turn will aid in getting $U_k(x, \tau)$ from (A10). The procedure to get $J(x, \tau)$ is very similar to the one used to get (A5).

Differentiating (A9) with respect to τ and using the same principle indicated in Step 1, one gets

$$J'(x, \tau) = \Phi(x, \tau) J(\tau, \tau) \tag{A10}$$

Step 5: In order to get $J(\tau, \tau)$, one goes back to (A9), from which

$$J(\tau, \tau) = (1 - \tau) + \int_0^\tau K(x, \tau) J(\xi, \tau)\, d\xi; \tag{A11}$$

$$= (1 - \tau) + \int_0^\tau x(1 - \xi) J(\xi, \tau)\, d\xi \tag{A12}$$

$$= (1 - \tau) + x \cdot r(\tau) \tag{A13}$$

where $r(\tau)$ is defined by

$$r(\tau) \triangleq \int_0^\tau (1 - \xi) J(\xi, \tau)\, d\xi; \quad 0 \leq \tau \tag{A14}$$

Step 6: The value of $r(\tau)$ can be determined by using, once again, a procedure similar to that used in Step 2. Differentiating (A14) with respect to τ

$$r'(\tau) = (1 - \tau) J(\tau, \tau) + \int_0^\tau (1 - \xi) J'(\xi, \tau)\, d\xi, \tag{A15}$$

Substituting (A10) in (A15)

$$r'(\tau) = (1 - \tau) J(\tau, \tau) + \int_0^\tau (1 - \xi) \Phi(\xi, \tau) J(\tau, \tau)\, d\xi \tag{A16}$$

The value of $\Phi(\xi, \tau)$ from (A8) can now be substituted in (A16).

$$r'(\tau) = (1 - \tau) J(\tau, \tau) + \tau \int_0^\tau (1 - \xi) J(\xi, \tau) J(\tau, \tau)\, d\xi \tag{A17}$$

using the definition of $r(\tau)$ from (A14)

$$r'(\tau) = (1 - \tau) J(\tau, \tau) + \tau J(\tau, \tau) r(\tau) = [G(\tau)]^2 \tag{A18}$$

where $G(\tau)$ is defined in (12)

Thus, the differential equation for $r(\tau)$ is obtained. The initial conditions for this differential equation can be obtained readily from (A14) as

$$r(\tau = 0) = r(0) = 0,$$

The procedure to obtain other equations in the text is similar. A more rigorous and elaborate proof can be found in Reference 10.

An application of Hockney's method for solving Poisson's equation

by R. COLONY and R. R. REYNOLDS

The Boeing Company
Seattle, Washington

INTRODUCTION

The classical techniques of separation of variables and eigenfunction expansions apply to a wide variety of boundary value problems in partial differential equations. Analogous procedures exist for certain partial difference equations that arise from discretization of the differential equations. The coefficients in the expansions associated with difference equations are defined by finite sums involving the eigenfunctions and the unknown function. Under the conditions specified in the next section, the coefficients for a discretized form of Poisson's equation satisfy a tridiagonal system of linear algebraic equations which are easily solved.

Hockney[5] has detailed a direct method for solving the five-point difference equation for Poisson's equation $\nabla^2\varphi = \rho$ when boundary conditions are periodic. We have worked out the complete development when the values of φ are given on the boundary of a rectangle (Dirichlet Problem) and the finite difference network is composed of rectangles. The restrictions illustrated in Figure 1 are removed in a later section simply by redefining ρ. The paper is self-contained and thus repeats some of Hockney's formulas.

PROBLEM

Consider the boundary value problem for the two dimensional Poisson equation

$$\frac{\partial^2\varphi(x,y)}{\partial x^2} + \frac{\partial^2\varphi(x,y)}{\partial y^2} \equiv \nabla^2\varphi(x,y) = \rho(x,y). \quad (1)$$

By the interior Dirichlet problem associated with (1), we mean the following. If $f(x, y)$ is a continuous function prescribed on the boundary B of some finite closed region R, and $\rho(x, y)$ is a continuous function prescribed in R, then determine a twice differentiable function $\varphi(x, y)$ such that

$$\nabla^2\varphi(x,y) = \rho(x,y), \qquad (x,y) \in R - B,$$
$$\varphi(x,y) = f(x,y), \qquad (x,y) \in B.$$

The rectangle oriented on a coordinate system as in Figure 1 is indicated by the coordinates of the corners. The point $(x, y) \in B$ if (x, y) is on the perimeter of the rectangle. The point $(x, y) \in R$ if (x, y) is inside the rectangle or if $(x, y) \in B$. The solution of the Dirichlet problem to be described here is restricted to the rectangle shown in Figure 1.

A method involving Fourier series may be used to great advantage if certain restrictions are placed on $f(x, y)$. These restrictions are

$$\left.\begin{aligned}\varphi(x,0) &= b_0(x)\\ \varphi(x,m) &= b_m(x)\end{aligned}\right\} 0 \le x \le l, \qquad \left.\begin{aligned}\varphi(0,y) &= 0\\ \varphi(l,y) &= 0\end{aligned}\right\} 0 \le y \le m. \quad (2)$$

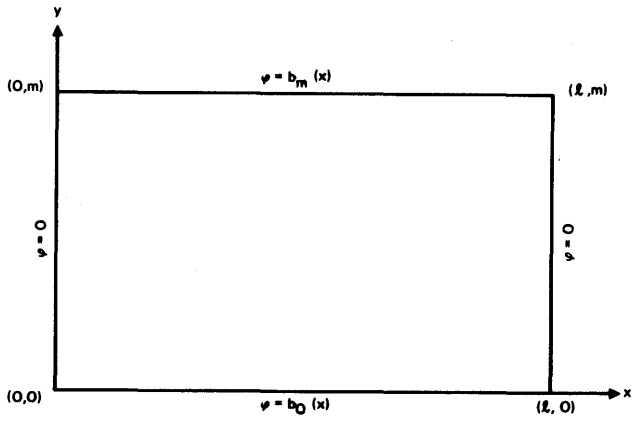


Figure 1

Now a separation of variables is employed and the solution to equation (1) is

$$\varphi(x, y) = \sum_{k=1}^{\infty} \bar{\varphi}_k(y) \sin \frac{\pi k x}{l},$$

$$\bar{\varphi}_k(y) = \frac{2}{l} \int_0^l \varphi(x, y) \sin \frac{\pi k x}{l} \, dx. \tag{3a}$$

$$\nabla^2 \sum_{k=1}^{\infty} \bar{\varphi}_k(y) \sin \frac{\pi k x}{l} = \rho(x, y), \tag{3b}$$

$$\sum_{k=1}^{\infty} \left\{ \frac{\partial^2}{\partial y^2} \bar{\varphi}_k(y) \sin \frac{\pi k x}{l} - \left(\frac{\pi k}{l}\right)^2 \bar{\varphi}_k(y) \sin \frac{\pi k x}{l} \right\} = \rho(x, y) \tag{3c}$$

$$\frac{\partial^2}{\partial y^2} \bar{\varphi}_k(y) - \left(\frac{\pi k}{l}\right)^2 \bar{\varphi}_k(y) = \bar{\rho}_k(y),$$

$$\bar{\rho}_k(y) = \frac{2}{l} \int_0^l \rho(x, y) \sin \frac{\pi k x}{l} \, dx. \tag{3d}$$

Equations of this type provide the motivation for considering analogous equations in the determination of a discrete approximation to φ.

METHOD

Finite difference representation

Rather than solve for $\varphi(x, y)$, it is usual to define the set of network lines

$$x_i = ih_x \quad (i = 0, 1, 2, \cdots N_x), \qquad h_x N_x = l,$$

$$y_j = jh_y \quad (j = 0, 1, 2, \cdots N_y), \qquad h_y N_y = m,$$

and solve for the $(N_x - 1)(N_y - 1)$ unknowns $\varphi_{i,j}$ that are approximations to $\varphi(x_i, y_j)$, $i = 1, 2, 3, \cdots N_x - 1$; $j = 1, 2, 3, \cdots N_y - 1$. As a matter of notation let

$$\rho_{i,j} = \rho(x_i, y_j).$$

The Laplacian operator in equation (1) is now replaced by the standard 5-point linear difference operator, and equation (1) is approximated as

$$\frac{\varphi_{i-1,j} - 2\varphi_{i,j} + \varphi_{i+1,j}}{h_x^2} + \frac{\varphi_{i,j-1} - 2\varphi_{i,j} + \varphi_{i,j+1}}{h_y^2} = \rho_{i,j},$$

$$i = 1, 2, 3, \cdots, N_x - 1; \quad j = 1, 2, 3, \cdots, N_y - 1. \tag{4a}$$

It is convenient to introduce the quantities

$$r^2 = h_y^2/h_x^2 \quad \text{and} \quad R = -2(1 + r^2)/r^2$$

and rewrite equation (4a) as

$$\varphi_{i,j-1} + r^2(\varphi_{i-1,j} + R\varphi_{i,j} + \varphi_{i+1,j}) + \varphi_{i,j+1} = h_y^2 \rho_{i,j} \equiv q_{i,j}, \quad i = 1, 2, 3, \cdots, N_x - 1; \quad j = 1, 2, 3, \cdots, N_y - 1. \tag{4b}$$

The boundary conditions enter in a straightforward manner—

$$\varphi_{i,0} = b_0(x_i), \quad \varphi_{i,N_y} = b_m(x_i), \quad \varphi_{0,j} = \varphi_{N_x,j} = 0. \tag{4c}$$

A brief look at equation (4b) as a partitioned matrix equation will be helpful:

$$\begin{bmatrix} A & I & 0 & \cdots & 0 \\ I & A & I & & \vdots \\ 0 & I & A & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & A \end{bmatrix} \cdot \begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \vdots \\ \varphi_{N_y-1} \end{bmatrix} = \begin{bmatrix} q_1 - \varphi_0 \\ q_2 \\ q_3 \\ \vdots \\ q_{N_y-1} - \varphi_{N_y} \end{bmatrix}, \tag{5a}$$

where

$$r^{-2}A = \begin{bmatrix} R & 1 & 0 & \cdots & 0 \\ 1 & R & 1 & & \vdots \\ 0 & 1 & R & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & R \end{bmatrix}, \tag{5b}$$

$$\varphi_j = \begin{bmatrix} \varphi_{1,j} \\ \varphi_{2,j} \\ \varphi_{3,j} \\ \cdot \\ \cdot \\ \cdot \\ \varphi_{N_x-3,j} \\ \varphi_{N_x-2,j} \\ \varphi_{N_x-1,j} \end{bmatrix} \qquad q_j = \begin{bmatrix} q_{1,j} \\ q_{2,j} \\ q_{3,j} \\ \cdot \\ \cdot \\ \cdot \\ q_{N_x-3,j} \\ q_{N_x-2,j} \\ q_{N_x-1,j} \end{bmatrix}$$

$$j = 1, 2, 3, \cdots, N_y - 1,$$

$$\varphi_0 = \begin{bmatrix} f(x_1, 0) \\ f(x_2, 0) \\ f(x_3, 0) \\ \cdot \\ \cdot \\ \cdot \\ f(x_{N_x-3}, 0) \\ f(x_{N_x-2}, 0) \\ f(x_{N_x-1}, 0) \end{bmatrix}, \quad \varphi_{N_y} = \begin{bmatrix} f(x_1, m) \\ f(x_2, m) \\ f(x_3, m) \\ \cdot \\ \cdot \\ \cdot \\ f(x_{N_x-3}, m) \\ f(x_{N_x-2}, m) \\ f(x_{N_x-1}, m) \end{bmatrix}, \tag{5c}$$

and I is the identity matrix of order $N_x - 1$.

As a future aid in reducing the amount of Fourier analysis, a rather simple algorithm, to be known as odd/even reduction, will be applied. This algorithm requires that N_y be an even integer, but this is generally a very mild restriction. Looking at equation (5a) as a system of matrix equations, we have

$$\begin{aligned} \varphi_{j-2} + A\varphi_{j-1} + \varphi_j &= q_{j-1}, \\ \varphi_{j-1} + A\varphi_j + \varphi_{j+1} &= q_j, \\ \varphi_j + A\varphi_{j+1} + \varphi_{j+2} &= q_{j+1}, \end{aligned} \tag{5d}$$

for $j = 2, 4, 6, \cdots, N_y - 2$. Pre-multiplying the jth equation by $-A$ and adding the $(j-1)$th and $(j+1)$th equations, we obtain the set of reduced equations

$$\varphi_{j-2} + A^*\varphi_j + \varphi_{j+2} = q_j^*, \tag{6a}$$

where

$$A^* = 2I - A^2, \qquad q_j^* = q_{j-1} - Aq_j + q_{j+1}.$$

Equation (6a) now represents the following sets of linear equations:

(i) for $i = 2, 3, 4, \cdots, N_x - 2$,

$$\begin{aligned} &\varphi_{i,j-2} - r^4\{\varphi_{i-2,j} + 2R\varphi_{i-1,j} + [R^2 + 2 - 2r^{-4}]\varphi_{i,j} \\ &\quad + 2R\varphi_{i+1,j} + \varphi_{i+2,j}\} + \varphi_{i,j+2} = q_{i,j}^*, \end{aligned} \tag{6b}$$

where

$$q_{i,j}^* = q_{i,j-1} - r^2\{q_{i-1,j} + Rq_{i,j} + q_{i+1,j}\} + q_{i,j+1};$$

(ii) for $i = 1$, $\varphi_{0,j} = 0$,

$$\begin{aligned} &\varphi_{1,j-2} - r^4\{[R^2 + 1 - 2r^{-4}]\varphi_{1,j} + 2R\varphi_{2,j} + \varphi_{3,j}\} \\ &\qquad + \varphi_{1,j+2} = q_{1,j}^*, \end{aligned} \tag{6c}$$

where

$$q_{1,j}^* = q_{1,j-1} - r^2\{Rq_{1,j} + q_{2,j}\} + q_{1,j+1};$$

(iii) for $i = N_x - 1$, $\varphi_{N_x,j} = 0$,

$$\begin{aligned} &\varphi_{N_x-1,j-2} - r^4\{\varphi_{N_x-3,j} + 2R\varphi_{N_x-2,j} \\ &\quad + [R^2 + 1 - 2r^{-4}]\varphi_{N_x-1,j}\} \\ &\qquad + \varphi_{N_x-1,j+2} = q_{N_x-1,j}^*, \end{aligned} \tag{6d}$$

where

$$q_{N_x-1,j}^* = q_{N_x-1,j-1} - r^2\{q_{N_x-2,j} + Rq_{N_x-1,j}\} + q_{N_x-1,j+1}.$$

In equations (6a)–(6d), $j = 2, 4, 6, \cdots, N_y - 2$.

Fourier transform

The $\varphi_{i,j}$ can be expressed by the finite Fourier series

$$\varphi_{i,j} = \sum_{k=1}^{N_x-1} \bar{\varphi}_{k,j} \sin(\pi ki/N_x), \tag{7a}$$

where the coefficients $\bar{\varphi}_{k,j}$ are determined from the following considerations. Multiply (7a) by $\sin(\pi ki/N_x)$, sum with respect to i, and change the order of summation:

$$(\alpha) \quad \sum_{i=0}^{N_x-1} \varphi_{i,j} \sin(\pi li/N_x) = \sum_{k=1}^{N_x-1} \bar{\varphi}_{k,j} \left[\sum_{i=0}^{N_x-1} \sin(\pi ki/N_x) \sin(\pi li/N_x) \right],$$

where the bracketed sum may be written

$$(\beta) \quad \frac{1}{2} \sum_{i=0}^{N_x-1} \cos(\pi(k-l)i/N_x) - \frac{1}{2} \sum_{i=0}^{N_x-1} \cos(\pi(k+l)i/N_x).$$

When $k = l$, the first of these sums is simply $N_x/2$; otherwise, it is expressed as the real part (R) of a

geometric series of exponentials

$$\frac{1}{2}R\left\{\sum_{i=0}^{N_x-1} e^{\iota\pi(k-l)\,i/N_x}\right\}$$

$$= \frac{1}{2}\,(1 - (-1)^{k-l})R\left\{\frac{1}{1 - e^{\iota\pi(k-l)/N_x}}\right\}$$

$$\cdot\,(1 - (-1)^{k-l})/4,$$

where ι designates the imaginary unit ($\iota^2 = -1$). Similarly, the second sum is $(1 - (-1)^{k+l})/4$, and the total expression (β) is either $N_x/2$ (for $k = l$) or 0 (for $k \neq l$). The right side of (α) thus collapses to $(N_x/2)\bar{\varphi}_{l,j}$; thus

$$\bar{\varphi}_{k,j} = (2/N_x)\sum_{i=1}^{N_x-1} \varphi_{i,j} \sin\,(\pi ki/N_x). \qquad (7b)$$

Furthermore, let

$$q_{i,j}^* = \sum_{k=1}^{N_x-1} \bar{q}_{k,j}^* \sin\,(\pi ki/N_x), \qquad (8a)$$

with

$$\bar{q}_{k,j}^* = (2/N_x)\sum_{i=1}^{N_x-1} q_{i,j}^* \sin\,(\pi ki/N_x). \qquad (8b)$$

Substituting equations (7a) and (8a) into equation (6b) and employing some trigometric identities give

$$(\gamma)\quad 0 = \sum_{k=1}^{N_x-1} \{\bar{\varphi}_{k,j-2} + \lambda_k\bar{\varphi}_{k,j} + \bar{\varphi}_{k,j+2} - \bar{q}_{k,j}^*\}$$

$$\cdot \sin\,(\pi ki/N_x),$$

where

$$\lambda_k = 2 - r^4\left(R^2 + 2 + 4R\cos\frac{\pi k}{N_x} + 2\cos\frac{2\pi k}{N_x}\right). \qquad (9)$$

Since (γ) is the Fourier expansion of the function zero, all coefficients are zero, and

$$\bar{\varphi}_{k,j-2} + \lambda_k\bar{\varphi}_{k,j} + \bar{\varphi}_{k,j+2} = \bar{q}_{k,j}^*,$$

$$j = 2, 4, 6, \cdots, N_y - 2;$$

$$k = 1, 2, 3, \cdots, N_x - 1. \qquad (10)$$

It is this set of $(N_y/2) - 1$ linear equations which poses the next problem.

Direct solution for even lines

With a Fourier transform of the boundary conditions φ_0 and φ_{N_y}, equation (10) is written in matrix form as

$$\Lambda_k\bar{\varphi}_k = \bar{q}_k^*, \qquad (11)$$

where

$$\Lambda_k = \begin{bmatrix} \lambda_k & 1 & 0 & \cdot & \cdot & \cdot & \cdot & 0 \\ 1 & \lambda_k & 1 & & & & & \cdot \\ 0 & 1 & \lambda_k & & & & & \cdot \\ \cdot & & & \cdot & & & & \cdot \\ \cdot & & & & \cdot & & & \cdot \\ \cdot & & & & & \cdot & & \cdot \\ \cdot & & & & & \lambda_k & 1 & 0 \\ & & & & & 1 & \lambda_k & 1 \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 & \lambda_k \end{bmatrix}, \qquad (12a)$$

$$\bar{\varphi}_k = \begin{bmatrix} \bar{\varphi}_{k,2} \\ \bar{\varphi}_{k,4} \\ \bar{\varphi}_{k,6} \\ \cdot \\ \cdot \\ \cdot \\ \bar{\varphi}_{k,N_y-6} \\ \bar{\varphi}_{k,N_y-4} \\ \bar{\varphi}_{k,N_y-2} \end{bmatrix} \qquad \bar{q}_k^* = \begin{bmatrix} \bar{q}_{k,2}^* - \bar{\varphi}_{k,0} \\ \bar{q}_{k,4}^* \\ \bar{q}_{k,6}^* \\ \cdot \\ \cdot \\ \cdot \\ \bar{q}_{k,N_y-6}^* \\ \bar{q}^k{}_{,N_y-4}^* \\ \bar{q}_{k,N_y-2}^* - \bar{\varphi}_{k,N_y} \end{bmatrix} \qquad (12b)$$

In equations (11) and (12) $k = 1, 2, 3, \cdots, N_x - 1$. The matrix Λ_k is factored as

$$\Lambda_k = L_kU_k \qquad (13)$$

such that

$$L_k = \begin{bmatrix} 1 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ -t_2 & 1 & 0 & & & & & & & \cdot \\ 0 & -t_4 & 1 & & & & & & & \cdot \\ \cdot & & & \cdot & & & & & & \cdot \\ \cdot & & & & \cdot & & & & & \cdot \\ \cdot & & & & & \cdot & & & & \cdot \\ \cdot & & & & & & 1 & 0 & & 0 \\ & \cdot & & & & & -t_{N_y-6} & 1 & & 0 \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & & -t_{N_y-4} & & 1 \end{bmatrix} \qquad (14a)$$

and

$$U_k = \begin{bmatrix} -t_2^{-1} & 1 & 0 & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & -t_4^{-1} & 1 & & & & & \cdot \\ 0 & 0 & -t_6^{-1} & & & & & \cdot \\ \cdot & & & \cdot & & & & \cdot \\ \cdot & & & & \cdot & & & \cdot \\ \cdot & & & & & \cdot & & \cdot \\ \cdot & & & & & -t_{N_y-6}^{-1} & 1 & 0 \\ \cdot & & & & & 0 & -t_{N_y-4}^{-1} & 1 \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 0 & -t_{N_y-2}^{-1} \end{bmatrix} \tag{14b}$$

where t_j is used as an abbreviation for $t_{k,j}$. From (13)

$$\lambda_k = -t_{k,2}^{-1},$$

$$\lambda_k = -t_{k,}^{-1}{}_j - t_{k,j-2}, \qquad j = 4, 6, 8, \cdots, N_y - 2.$$

The elements $t_{k,j}$ then are given as

$$t_{k,2} = -1/\lambda_k,$$

$$t_{k,j} = -1/(t_{k,j-2} + \lambda_k), j = 4, 6, 8, \cdots, N_y - 2. \tag{15}$$

Now equation (11) is written as

$$L_k U_k \bar{\varphi}_k = \bar{q}_k{}^*. \tag{16a}$$

By introducing a new set of vectors

$$s_k = U_k \bar{\varphi}_k, \tag{16b}$$

the simple form of

$$L_k s_k = \bar{q}_k{}^* \tag{16c}$$

is obtained.

The vector s_k is obtained from equation (16c) by simple forward substitution

$$s_{k,2} = \bar{q}_{k,2}{}^*,$$

$$s_{k,j} = \bar{q}_{k,j}{}^* + t_{j-2} s_{k,j-2}, j = 4, 6, 8, \cdots, N_y - 2. \tag{17}$$

Now the vector $\bar{\varphi}_k$ is obtained from equation (16b) by simple back substitution

$$\bar{\varphi}_{k,N_y-2} = -t_{N_y-2} s_{k,N_y-2},$$

$$\bar{\varphi}_{k,j} = t_j(\bar{\varphi}_{k,j+2} - s_{k,j}), j = N_y - 4, N_y - 6, \cdots, 2. \tag{18}$$

After solving equation (18), note that $\bar{\varphi}_{k,j}$ has been determined for $k = 1, 2, 3, \cdots, N_x - 1$ and $j = 2, 4, 6, \cdots, N_y - 2$. The $\bar{\varphi}_{k,j}$ are the solutions of the Fourier transform of equation (6b), which has subscripts $i = 2, 3, 4, \cdots, N_x - 2$ and $j = 2, 4, 6, \cdots, N_y - 2$. Now the Fourier synthesis is performed using equation (7a), and $\varphi_{i,j}$ is determined for $i = 2, 3, 4, \cdots, N_x - 2$ and $j = 2, 4, 6, \cdots, N_y - 2$. Equations (6c) and (6d) are rearranged such that

$$\varphi_{1,j-2} - [r^4(R^2 + 1) - 2]\varphi_{1,j} + \varphi_{1,j+2} = q_{1,j}{}^{**},$$

where

$$q_{1,j}{}^{**} = q_{1,j}{}^* + r^4[2R\varphi_{2,j} + \varphi_{3,j}];$$

and

$$\varphi_{N_x-1,j-2} - [r^4(R^2 + 1) - 2]\varphi_{N_x-1,j} + \varphi_{N_x-1,j+2} = q_{N_x-1,j}{}^{**},$$

where

$$q_{N_x-1,j}{}^{**} = q_{N_x-1,j}{}^* + r^4[2R\varphi_{N_x-2,j} + \varphi_{N_x-3,j}],$$

$$j = 2, 4, 6, \cdots, N_y - 2.$$

The similarity of the above equations and equation (10) is shown below

$$\bar{\varphi}_{k,j-2} + \lambda_k \bar{\varphi}_{k,j} + \bar{\varphi}_{k,j+2} = \bar{q}_{k,j}{}^*,$$

$$\varphi_{1,j-2} + \gamma\varphi_{1,j} + \varphi_{1,j+2} = q_{1,j}{}^{**},$$

$$\varphi_{N_x-1,j-2} + \gamma\varphi_{N_x-1,j} + \varphi_{N_x-1,j+2} = q_{N_x-1,j}{}^{**},$$

where

$$\gamma = -[r^4(R^2 + 1) - 2].$$

The above two equations are again written in matrix form given in equation (11) but with γ replacing λ_k.

At this step, half of the unknown $\varphi_{i,j}$ are determined.

Direct solution for odd lines

Determination of the balance of the unknown $\varphi_{i,j}$ is a much more simple task. For j an odd integer, equation (4b) becomes

$$r^2(\varphi_{i-1,j} + R\varphi_{i,j} + \varphi_{i+1,j}) = h_y{}^2\rho_{i,j} - \varphi_{i,j-1} - \varphi_{i,j+1} \equiv b_{i,j},$$

$$i = 1, 2, 3, \cdots, N_x - 1;$$

$$j = 1, 3, 5, \cdots, N_y - 1. \tag{19}$$

With boundary conditions $\varphi_{0,j} = \varphi_{N_x,j} = 0$, the above equation is written in matrix form as $A\varphi_j = b_j$, or

$$r^{-2}A\varphi_j = r^{-2}b_j \equiv b_j{}^*, \tag{20}$$

where

$$b_j^* = \begin{bmatrix} b_{1,j}^* \\ b_{2,j}^* \\ b_{3,j}^* \\ \cdot \\ \cdot \\ \cdot \\ b_{N_x-3,j}^* \\ b_{N_x-2,j}^* \\ b_{N_x-1,j}^* \end{bmatrix}$$

and $r^{-2}A$ and φ_j were previously defined in (5b) and (5c). The matrix $r^{-2}A$ is factored into $r^{-2}A = LU$, where L is lower triangular and U is upper triangular:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdot & 0 \\ -u_1 & 1 & 0 & & \cdot \\ 0 & -u_2 & 1 & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & 1 & 0 & 0 \\ \cdot & & & & \\ \cdot & & -u_{N_x-3} & 1 & 0 \\ \cdot & & & & \\ 0 & \cdot & 0 & -u_{N_x-2} & 1 \end{bmatrix}, \qquad (21a)$$

$$U = \begin{bmatrix} -u_1^{-1} & 1 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & -u_2^{-1} & 1 & & & & \cdot \\ 0 & 0 & -u_3^{-1} & & & & \cdot \\ \cdot & & & \cdot & & & \cdot \\ \cdot & & & & \cdot & & \cdot \\ \cdot & & & & & & \cdot \\ \cdot & & & & & -u_{N_x-3}^{-1} & 1 & 0 \\ \cdot & & & & & & & \\ \cdot & \cdot & \cdot & \cdot & 0 & -u_{N_x-2}^{-1} & 1 \\ 0 & & & & 0 & 1 & -u_{N_x-1}^{-1} \end{bmatrix} \qquad (21b)$$

where the elements u_i are given as

$$u_1 = -1/R,$$

$$u_i = -1/(u_{i-1} + R), \quad i = 2, 3, 4, \cdots, N_x - 1. \qquad (22)$$

Now equation (20) is written as

$$LU\varphi_j = b_j^*. \qquad (23a)$$

By introducing the new set of vectors

$$v_j = U\varphi_j, \qquad (23b)$$

the simple form of

$$Lv_j = b_j^* \qquad (23c)$$

is obtained.

The vector v_j is obtained from equation (23c) by simple forward substitution

$$v_{1,j} = b_{1,j}^*,$$

$$v_{1,j} = b_{i,j}^* + u_{i-1}v_{i-1,j}, \quad i = 2, 3, 4, \cdots, N_x - 1. \qquad (24)$$

Now the vector φ_j is obtained from equation (23b) by simple back substitution

$$\varphi_{N_x-1,j} = -u_{N_x-1}v_{Nx-1,j},$$

$$\varphi_{i,j} = u_i(\varphi_{i+1,j} - v_{i,j}),$$

$$i = N_x - 2, N_x - 3, \cdots, 1. \qquad (25)$$

Extension of boundary conditions

Let

$$\nabla^2\varphi = \rho$$

in the rectangle (Figure 1) with

$$\varphi(0, y) = f(y) \quad \text{and} \quad \varphi(l, y) = g(y), \quad 0 \leq y \leq m.$$

and let

$$\varphi = \varphi_A + f(y)\frac{l - x}{l} + g(y)\frac{x}{l};$$

then

$$\nabla^2\varphi = \nabla^2\varphi_A + \frac{d^2f(y)}{dy^2}\frac{l-x}{l} + \frac{d^2g(y)}{dy^2}\frac{x}{l} = \rho,$$

$$\nabla^2\varphi_A = \rho - \frac{d^2f(y)}{dy^2}\frac{l-x}{l} - \frac{d^2g(y)}{dy^2}\frac{x}{l} \equiv \rho_A.$$

Now

$$\nabla^2\varphi_A = \rho_A, \qquad \varphi_A(0, y) = \varphi_A(l, y) = 0,$$

is directly solvable by the methods described in this paper.

Extensions to the method

The preceding portions of this section employed many conditions which allowed the method to be more clearly explained. Many of these conditions can be removed with no more serious result than changing a few coefficients in some of the equations. The following extensions do not alter the basic algorithm:

a) $\dfrac{d\varphi}{dx}(0, y) = \dfrac{d\varphi}{dx}(l, y) = 0, \qquad 0 \le y \le m,$

b) general boundary conditions of the second or third kind at $(x, 0)$ and (x, m), $0 \le x \le l$,
c) exterior problems with asymptotic boundary conditions,
d) periodic boundary conditions,
e) more general elliptic equations such as

$$\frac{\partial}{\partial x}\left(a(y)\frac{\partial\varphi}{\partial x}\right) + \frac{\partial}{\partial y}\left(b(y)\frac{\partial\varphi}{\partial y}\right) + c(y)\varphi = \rho(x, y),$$

f) applications to one dimensional wave equations associated with cyclic processes,
g) conformal transformations to extend the types of boundaries which may be used.

A large class of problems exist where this method loses its directness but still may be applicable:

a) successive approximation of the non-linear equations resulting from discretizing certain non-linear elliptic equations,
b) solving blocks of difference equations in the process of some block iteration techniques,
c) solving Poisson's equation over non-rectangular regions.

APPLICATION

The equations of motion of a viscous, unsteady, incompressible fluid past a finite flat plate are written in the elliptic coordinate system as

$$\left(\frac{1}{q^2}\frac{\partial}{\partial t} - \frac{1}{Re}\nabla^2\right)\zeta = -u\frac{\partial\zeta}{\partial\xi} - w\frac{\partial\zeta}{\partial\eta}, \qquad (26a)$$

$$\nabla^2\psi = -\zeta/q^2, \qquad \text{where} \qquad \nabla^2 \equiv \frac{\partial^2}{\partial\xi^2} + \frac{\partial^2}{\partial\eta^2}. \qquad (26b)$$

The vorticity is denoted by ζ and the stream function by ψ; $q = \partial(x, y)/\partial(\xi, \eta)$ is the modulus of transformation between the cartesian coordinates (x, y) and the elliptic coordinates (ξ, η). The components of velocity u and w are defined as

$$u \equiv \frac{\partial\psi}{\partial\xi}, \qquad -w \equiv \frac{\partial\psi}{\partial\eta}.$$

The domain is rectangular and is given by

$$0 \le \xi \le E, \qquad 0 \le \eta \le \pi.$$

The vorticity equation and the stream function equation are coupled together in a non-linear manner. One method of solution which has had success is shown by the following algorithm:

a) estimate ψ,
b) calculate u, w from estimate of ψ,
c) solve linearized vorticity equation (26a) for ζ,
d) solve (26b) for stream function with newly obtained value of ζ,
e) repeat from b) with new ψ until convergence.

The boundary conditions on the stream function equation are

$$\psi(0, \eta) = \psi(\xi, 0) = \psi(\xi, \pi) = 0,$$

$$\psi(E, \eta) = f(\eta).$$

The direct solution of the stream function equation makes the above algorithm attractive from the standpoint of computer efficiency.

For $N_x = 64$ and $N_y = 46$ the execution time for one direct solution of equation (26b) was .67 sec.* Prior to implementation of the direct solution, equation (26b) was solved by successive-line-over-relaxation (SLOR). Solving equations (26a, b) for 150 time steps typically took about 90 minutes when SLOR was used. This execution time was reduced to 15 minutes using the direct methods shown in this paper.

* Execution time is for the UNIVAC 1108 computer with single precision arithmetic. The Fast Fourier Transform Algorithm[3,4] was used to evaluate the finite sine series.

BIBLIOGRAPHY

1 B L BUZBEE G H GOLUB C W NIELSON
The method of odd/even reduction and factorization with application to Poisson's equation
Stanford University Computer Science Department Tech Report No CS 128 1969

2 L COLLATZ
The numerical treatment of differential equations
Third edition Springer 1960

3 J W COOLEY J W TUKEY
An algorithm for the machine calculation of complex fourier series
Math Comp 19 pp 297–301 1965

4 J W COOLEY P A W LEWIS P D WELCH
The fast fourier transform algorithm and its applications
IBM Watson Research Center Yorktown Heights New York 1967

5 R W HOCKNEY
A fast direct solution of Poisson's equation using fourier analysis
J ACM Vol 12 pp 95–113 1965

6 R W HOCKNEY
The potential calculation
Conference on Numerical Simulation of Plasma Los Alamos 1968

7 R S VARGA
Matrix iterative analysis
Prentice–Hall 1962

Architecture of a real-time fast fourier radar signal processor

by ARTHUR S. ZUKIN and S. Y. WONG

Hughes Aircraft Company
Culver City, California

SUMMARY

This paper describes the architecture of an all-digital signal processor for a high pulse repetition frequency (high PRF) radar.[1,2] The processor replaces a bank of hundreds of (approximately) 100 Hz bandwidth analog filters with an equivalent but more capable and smaller, lighter and less expensive digital system. The digital system acts in real-time by converting input signals (time domain) from analog to digital form, collecting sets of such converted signals and then performing a discrete Fourier* transform[3,4] upon them. Thus, it produces a (frequency domain) result which is equivalent to the output from the bank of analog filters or from a spectrum analyzer.

Because the processor is employed on a full-time basis solely to perform the Fourier transform, it can be designed to do this task at lower cost than a general purpose computer and with lower performance logic circuits and memory than would be required in a general purpose computer. It can perform direct and inverse Fourier transforms and could be used in pattern matching and convolution.[5] The processor described can be modified and/or adapted to low or medium PRF[1,2,6,7] and/or synthetic array modes as well as to communications systems. For example, this basic design has been successfully used in communications systems to perform both the direct and inverse transform and serves as both demodulator and modulator.

The selected processor architecture separates the functions to be performed from each other and in most cases assigns physically distinguishable portions of the equipment to the functions because:

1. If the functions are not conceptually separated, the overall task is unduly complex. Like a conventional digital computer, the processor is a collection of simple blocks; though complex considered in the ensemble, they are easily understood separately.

2. Separating functions and associating them one-for-one with equipment demonstrates the equivalence between the generalized fast Fourier (Cooley-Tukey)[8–18] procedure and the equipment.

3. By designing for the application from the outset one can pick efficient hardware for implementing the functions. The result is a straightforward design with a comparatively small number of efficient functional blocks.

INTRODUCTION

System capability

The analog system which is equivalent to the digital signal processor would have a total bandwidth of 146 kHz. (See Table I.) This would be comprised of two 73 kHz subbands.* It would have two modes of operation.

In the first mode each subband would drive 512 filters having a matched bandwidth of 143 Hz. Thus, the system would provide a total bandwidth of 146 kHz in 1,024 143 Hz filters.

In the second mode, each subband would drive 1,024 filters having a matched (filter) bandwidth of 72 Hz. Thus, the system would provide a total bandwidth of 146 kHz in 2,048 72 Hz filters.

In the digital system, the sampling rate for both in-phase and quadrature signals* is 146,000 per second per subband. I.e., in each of the two subbands, there is a vector sampling rate of 146,000 per second and the total vector sampling rate is 292,000 per second which

* Fast Fourier transform digital processing and digital filters and their relationship are discussed in the appendix.

* See "Terminology Used."

TABLE I—Comparison of Analog and Digital Systems

	Mode 1	*Mode 2*
Analog System		
a) Total Bandwidth, Hz (2 subbands)	146,000	146,000
b) Bandwidth per Subband, Hz	73,000	73,000
c) No. of Filters per Subband	512	1,024
d) Filter Bandwidth, Hz	143	72
Digital System		
a) Total Bandwidth, Hz (2 subbands)	146,000	146,000
b) Bandwidth per Subband, Hz	73,000	73,000
Total Vector Sampling Rate per Sec	292,000	292,000
Vector Sampling Rate per Sec, per Subband	146,000	146,000
Frame Time, Millisec	7	14
No. of Vector Samples per Frame per Subband	1,024	2,048
Filters "Created" per Frame per Subband		
c) Filters "Saved" per Frame per Subband	512	1,024
d) Filter Bandwidth, Hz	143	72

is twice the bandwidth of the analog system and satisfies the Nyquist requirement.

In the first mode, the frame time (that is, the time for one set of samples) is 7 msec. which results in a filter width of $(0.007 \text{ sec})^{-1}$ or 143 Hz as in the equivalent analog system. During the 7 msec frame 1,024 vector samples are made for each of the two subbands. Thus, 1,024 data enter the processor for each subband even though only half this quantity (512) filters are to be created. This is necessary because of the 2:1 oversampling mentioned in the previous paragraph. At the end of the fast Fourier procedure the desired 512 "filters"* are retained and the other 512 are ignored. (If the fast Fourier procedure were done base-2, one could merely skip one-half of the last base-2 procedure.)

Likewise in the second digital mode, twice as many samples are taken as the desired number of retained filters. The quantities are 2,048 vector samples in a 14 msec frame per subband and 1,024 filters are retained for each subband. The filter width is $(0.014 \text{ sec})^{-1}$ or 72 Hz.

It is important to note that the provision of both 512 143 Hz filters and 1,024 72 Hz filters in the same analog system is very expensive and the likely compromise for the sort of system described here would be to supply only 512 filters which in mode one would be switched from one subband to the other. Another highly likely economy move would be the use of single compromise bandwidth somewhere between 72 and 143 Hz and use of the same 512 filters for both modes one and two. Despite its lower cost the digital system does not need to make either of these sacrifices. If enough memory is provided for the 1,024 filter per subband case and enough speed for the 7 msec frame case, the "variable" filter bandwidth is "free."

Radar background information[1,2,6,7,19]

Radar systems determine target range (R) by measuring time (t) between transmission of a pulse and reception of the target echo signal. Since the radar signal travels a distance of $2R$, one obtains

$$t = \frac{2R}{c} \quad \text{or} \quad R = \frac{tc}{2} \tag{1}$$

where c is the velocity of propagation, roughly 160,000 nautical miles per second. If the target is moving with respect to the radar the target velocity (v or $\dot{R}$) along the line of sight will cause the received echo signal to differ from the transmitted signal frequency (f) by an amount, Δf. This is the well known doppler effect.

$$\Delta f = (2v/c)f \tag{2}$$

Replacing c by λf (λ is the wavelength of the radar) gives

$$\Delta f = 2v/\lambda \tag{3}$$

For velocities of interest (up to 5000 feet per second) and wavelengths of interest (0.03 foot to 3 feet) the absolute value of the doppler frequency shift may be as great as 300,000 Hz and the frequency region of interest can easily span one-half megahertz.

Because the pulse radar is inherently a sampled data system, the spectrum[1,2,6,19] of the received signal is complex even in an idealized case. The spectrum of the transmitted signal consists of spectral lines separated from each other by the pulse repetition frequency (f_r); the envelope of the spectral lines has a $(\sin X)/X$ shape, is centered about the transmitted frequency (f) and has a frequency width of $2/\tau$ between the first pair of zero crossings where τ is the width of the transmitted pulse. The important attribute of the transmitted waveform is that it is comprised of f, $f \pm f_r$, $f \pm 2f_r$, $f \pm 3f_r$, and so on.

As a result, the received signal is not simply $f + \Delta F$ as would be surmised from Equation (2) but contains many frequencies, $f + \Delta F$, $f \pm f_r + \Delta f$, $f \pm 2f_r + \Delta f$, $f \pm 3f_r + \Delta f$ and so on. Therefore, doppler frequencies of F, $F + f_r$, $F + 2f_r$ etc. are indistinguishable from each other. Because of this the total doppler frequency bandwidth that can be measured unambiguously is f_r.

* See "Terminology Used."

The consequence is that in order to measure velocity unambiguously, f_r must be at least as great as the total doppler bandwidth and the pulse repetition period (PRP) which is the inverse of the PRF may perforce be of the order of a few microseconds or 10 microseconds. This means that the maximum range which can be measured unambiguously is very, very small. For example, if the PRF is 100,000 per second (PRP is 10 microseconds) the maximum unambiguous range which can be measured is about 0.8 nautical mile and all ranges will be measured modulo-0.8 nautical mile. On the other hand as long as the doppler frequencies span no more than 100 kHz, they can be unambiguously measured modulo-100 kHz. With the above parameters given, the system would be what is called a high PRF radar; that is, it would be a radar which measures range ambiguously but measures doppler unambiguously.

If the PRF were changed to 1000 per second, (PRP of 1000 microseconds), the unambiguous range would be 80 nautical miles and ranges would be measured modulo-80 nautical miles. But doppler frequencies could be measured unambiguously only as long as they spanned no more than 1 kHz and would be measured modulo-1 kHz. For most applications the doppler information would be essentially useless at this PRF. With these parameters, the system would be a low PRF radar for targets of less than 80 miles range. That is, range would always be measured unambiguously but velocities of practical interest would be measured ambiguously.

When the radar parameters are such that both range and velocity are measured ambiguously it is said to be a medium PRF radar. Although this appears to be a useless system, it is not necessarily so; with modern digital equipment both range and frequency (doppler) can be measured ambiguously with each of a set of related PRF's and the unambiguous values computed from the sets of measurements. For example, the combination of ambiguous range measurements modulo-19 nautical miles and modulo-10 nautical miles yields unambiguous range out to 190 nautical miles.

Terminology used

The system described represents an effort to apply new technology to an old problem. Moreover, the technology is digital and the problem is traditionally solved with analog implementation. Therefore, it appears appropriate to define a few terms which are not familiar to all.

Sample and hold circuits which are the source of the input signals to the processor are used in an analog-to-digital converter when it is desirable to make a measurement of a signal and to know precisely when the input signal corresponds to the results of the measurement.

In-phase and quadrature signals (I and Q) refer to the component of the input signal which is in phase with a reference signal and to the component which is in quadrature (leads or lags by 90 electrical degrees) with the in-phase signal. These correspond to the real and imaginery portions of the input signal which is assumed to be the vector sum of all of the input components.

Subbands refer to the division of the total input frequency band into two subbands each of which contains one-half of the total bandwidth of the system. This may be done for a number of reasons. For example, the input signal may have too large range between maximum and minimum amplitude for the analog-to-digital converter. In this case, and if the distribution of energy across the band is fairly uniform, the range may be reduced by the use of subbands each covering a portion of the spectrum.

Since an all-digital processor uses no physical filters, the equivalent of analog filter responses is synthesized by the processor using the time domain to frequency domain Fourier transform.* At the end of the transform procedure the input data has been converted to a set of vector numbers representing the voltages one would have obtained from a set of analog filters with the same characteristics as the digital filters. Each vector is called a *filter* or a *complex filter* and is subsequently multiplied by its complex conjugate to obtain a quantity representative of the power which would have passed through the equivalent analog filter in the same time interval.

SUMMARY SYSTEM DESCRIPTION

The digital processor block diagram is shown in Figure 1. The driving circuits (sample and hold) and

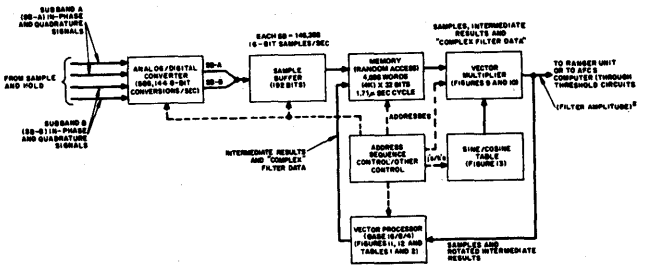


Figure 1—Digital processor block diagram

* See appendix.

and one of the three fast Fourier steps has been completed.

During the next fast Fourier step, the procedure is similar with alternate bursts of five or six memory cycles employed to read in new data and 16 memory cycle bursts employed to exchange memory data destined for the vector processor with processed data from the vector processor. There are, however, two important differences from the first step. These are:

1. In order to perform the required rotation, operands (complex) leaving the memory are multiplied by appropriate unit vectors[3,4,8–10] in the vector multiplier before entering the vector processor. For a rotation of θ radians, the unit vector $\cos\theta + i\sin\theta$, is generated by the sine/cosine table which is provided with the rotation angle by the address sequence control. Trigonometric functions of angles from zero to $\pi/4$ are generated directly by an interpolation procedure; functions for angles from $\pi/4$ to 2π are derived from the directly generated functions. The procedure and its mechanization are simple and do not require extensive equipment.

2. Operands are taken in groups of eight (G-8) for the second step of a 14-millisecond 4,096 sample frame and in groups of four (G-4) for the second step of a 7-millisecond 2,048 sample frame. However, the transfers between the memory and the vector processor still occur in bursts of 16 memory cycles. The 16 words involved represent two G-8s or four G-4s.

During the last of the three fast Fourier steps processing again utilizes G-16s. However, the last step differs from the first in that the complex data is multiplied by the appropriate unit vectors in the vector multiplier. The multiplication and sine/cosine procedures are identical to those employed in the second step.

The data returned to the memory during the third (last) step represents the complex magnitudes of the filtered input signals (filters) and enters the memory during the last one-third of the frame after that in which it was obtained. During the next frame, as new data enters the memory in bursts of five or six memory cycles, an equal quantity of filters is read out of the memory. Like all information leaving the memory, the filters pass through the vector multiplier where each complex filter is multiplied by its own complex conjugate to provide the square of its magnitude according to the well-known relationship,

$$\text{magnitude}^2 = f(a + ib) = a^2 + b^2 = (a + ib)(a - ib).$$

The squared magnitudes are proportional to the power output of the equivalent analog filters and are the digital processor output signals to the threshold circuits. In these circuits they are compared against a threshold level and an output is generated whenever the threshold is crossed.

OVERALL DESIGN DECISIONS

Some system design decisions and equipment choices and mechanizations are described in this section.

Memory selection

For a particular level of capability, probably the most meaningful measure of the worth of a digital processor design is the ability to "get by" with a single economical, easily produced memory with little penalty in other portions of the processor. The memory in this design uses 4,096 words of 32 bits with a cycle time of 1.71 microseconds. Thus, it stores 131,000 bits and requires an information rate of less than 19 bits per microsecond.

Because data for an entire 7- or 14-millisecond frame is processed as a "set," the absolute minimum amount of memory is that required to store the 4,096 samples obtained during the longer frame. If only this much storage is provided, processing must be performed instantaneously at the end of a data gathering period and results transferred to the using equipment instantaneously. These conditions are obviously impossible. A reasonable upper limit is the storage required for two frames of data or $2 \times 4{,}096 = 8{,}192$ samples. This results in a system in which data processing takes one frame time and occurs during the next frame after the data is gathered, and the processed data is unloaded during the second frame after its samples were loaded into the memory simultaneously with the loading of new data.

Serious effort was made to avoid using a memory large enough for two frames of data, but this was not

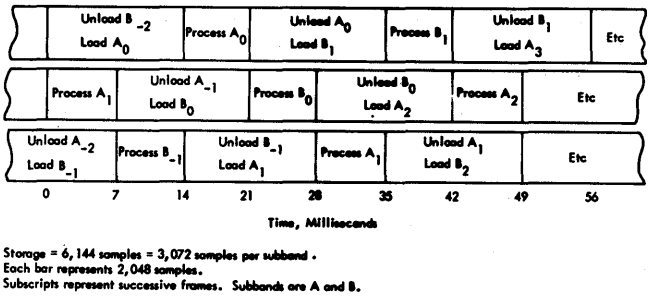


Figure 3—Staggered processing memory storage requirement and frame-timing for 14 milliseconds, 2 subbands, 2,048 samples per subband per frame

the receiving circuits (threshold circuits) are conventional and will not be considered here.

Data flow in the digital processor is designed to provide a simple sequence in which information is cycled between the memory and the vector processor until the (input) vector samples are converted to (output) vector quantities representing the real and imaginary portions of the signal voltage which has "passed through" an equivalent analog filter. Input data (Figure 1) flows into the memory and thence from memory through the vector multiplier and vector processor and back to the memory. This loop flow occurs three times, the information returned to the memory after the third step representing the filter outputs. These are then read out of the memory through the vector multiplier in which they are converted into the squares of the (scalar) magnitudes, and become the output of the digital processor.

In-phase and quadrature input signals for each of the two subbands are digitized in the analog/digital converter to provide 8-bit (seven bits plus sign) conversion of the in-phase and the quadrature signals. A total of 585,144 8-bit conversions occur each second; each set of four 8-bit conversions represents the in-phase and quadrature components (I and Q) of the signals of the two subbands and results in one 32-bit word comprised of two 16-bit halves; each half contains the I and Q for one subband. Thus, the average rate into the memory is one-quarter of 585,144 or 146,286 words per second.

Five or six input words are collected in the sample buffer and in a burst of five or six successive memory cycles (see Overall Design Decisions) are read into the memory at the same time that an equal number of filters* are read out of the memory. The procedure is an exchange of data in which data flowing into the memory replaces the data flowing out of the same memory location at a rate of one word per 1.71 microseconds memory cycle. Data leaving the memory represents the complex filter output voltage and is multiplied by its complex conjugate in the vector multiplier to form the square of the magnitude.

When the processor is first turned on, no useful processing can occur until one complete frame of data has entered the memory. Subsequently groups of 16 operands per subband (called G-16's**) are read out of the $2M$-word memory from locations (0, $M/16$, $2M/16$, $3M/16$, . . .), (1, $M/16 + 1$, $2M/16 + 1$, $3M/16 + 1$, . . .), (2, $M/16 + 2$, $2M/16 + 2$, $3M/16 +$ 2, . . .), and so on. The 16 words are read in a burst of 16 memory cycles while five or six words are collected in the input buffer. Between successive 16 word bursts, a burst of five or six memory cycles is employed to load five or six new data words and unload five or six filters.

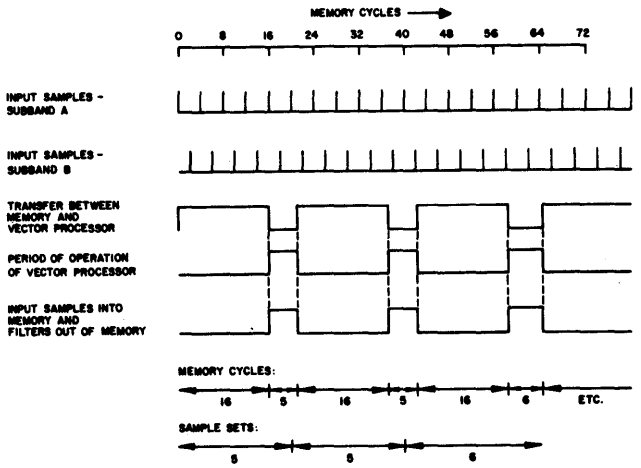


Figure 2—Timing diagram for burst processing

Like all data leaving the memory the G-16 passes through the vector multiplier word by word, but in this first step is passed through the vector multiplier without change and on into the vector processor. In the vector processor, the G-16 data for each of the subbands is subjected to a base-16[9] fast Fourier transform procedure. The time allowed for this is five or six memory cycles. (See timing diagrams, Figure 2.) At the end of the base-16 procedure, the 16 pairs of results*** in the 16 words of the G-16 are returned word by word to the memory in a burst of 16 memory cycles to replace the 16 words of the next G-16 as they are read out of the memory through the vector multiplier to the vector processor, etc.

Ideally, the processed G-16 would be returned to locations in the memory from which the (same) input G-16 had been obtained. This would require one memory cycle to read the data plus another memory cycle to write, resulting in a total of two memory cycles per word. However, the information *exchange* can be performed during a single memory cycle per word. The effectively double memory speed is not obtained without penalty and forces the use of an address sequence control (see Overall Design Decisions) to keep track of the changing memory addresses. After the last G-16 passes through the vector processor, it is returned to the location from which the first G-16 was obtained

* Which represent the processed result of data gathered two frames before the input frame.

** Each of the 16 words in the G-16 contains 16 bits of subband A data and 16 bits of subband B data.

*** One result per subband per word.

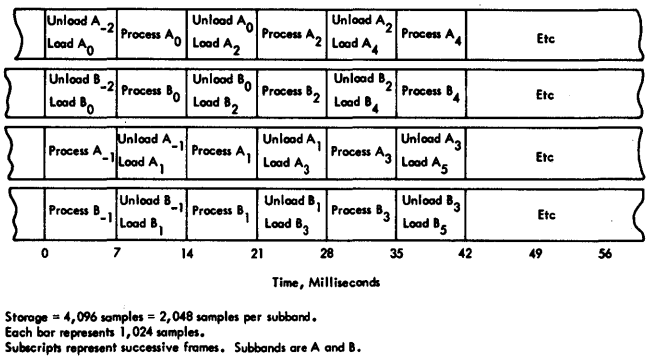


Figure 4—Memory storage requirement and frame-timing for 7 milliseconds, 2 subbands, 1,024 samples per subband per frame. Processing not staggered

successful. A scheme called "staggered processing" was devised in which the start of the frame times for the two subbands A and B, differed by 7 milliseconds (i.e., one-half of a 14-millisecond frame) and the information for the individual subband frame was processed during a period of 7 milliseconds. This is illustrated by Figure 3; the alternate of nonstaggered processing is illustrated for the 7- and 14-millisecond frame times by Figures 4 and 5. Although potentially useful, staggered processing was not adopted because of a number of difficulties, namely:

1. It appears desirable to store (at least) two samples or pieces of data in each memory word. If staggered processing is used, the two must be from the same subband. However, it is desirable that the two represent equivalent data from the (in this case) two subbands in order that the use of twice as many trigonometric

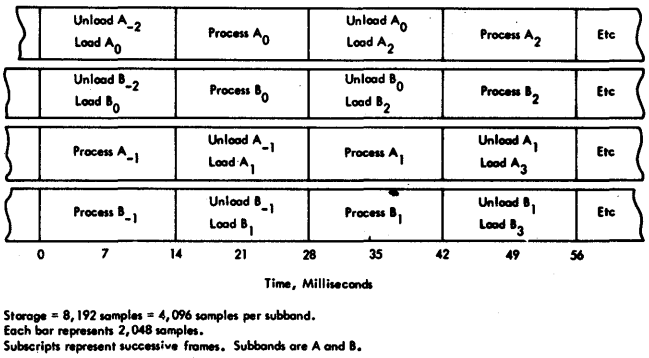


Figure 5—Memory storage requirement and frame-timing for 14 milliseconds, 2 subbands, 2,048 samples per subband per frame. Processing not staggered

function "look-ups" and twice as fast a trigonometric table be avoided.

2. A requirement for a special sorting pass of the filtered data to obtain the desired output sequence and/or the acceptance of n parallel streams, each of which is a properly sequenced stream containing $1/n$th of the results.

3. Greater complexity in the address sequence control because of the multiplicity of address patterns that accompany the staggered processing scheme.

Choice of arithmetic base of vector processor

The choice of arithmetic base[8,9] for the vector processor is closely related to selection of the memory. The choice of a particular processor base leads to a total memory input–output data rate and the choice of a memory data rate places a lower limit on the processor (arithmetic) base. Accompanying each base is a minimal number of registers within the vector processor; this is the product of the base times the number of samples per memory word. Thus, the higher the base, the more registers are needed in the vector processor itself. However, the higher base processor can be internally mechanized as a series of lower base substeps. Thus an externally viewed base-16 processor could internally use four base-2 or two base-4 substeps. If a higher base is used in this manner, the total amount of equipment, other than registers, is only weakly dependent on the base. In the (externally viewed) base-16 design described, fewer multipliers are used in the entire system than would have been required if a base-4 processor were used.

Moreover, the use of a high base is a definite advantage in reducing the speed of the memory and the arithmetic circuits. This is so because for N samples, the number of memory cycles and of multiplications per unit of time varies as $\log_B N$ where B is the arithmetic base. Since $\log_B N = \log_e N/\log_e B$, the required memory speed and arithmetic speed vary inversely with the natural logarithm of the base.

If the base is two or four, the vector operations within the processor are trivial; that is, they are multiplications by the sine and cosine of 0, $\pi/2$, π and $3\pi/4$ and thus no explicit multiplier is required within the vector processor.* Therefore, except under unusual circumstances, the lowest base to be considered should be four.

* As stated in Summary System Description and illustrated in the system block diagram (Figure 1), all vector multiplications other than those within the vector processor, and all rotation of vectors are performed by the vector multiplier through which all information flows as it leaves the memory.

In the case of the system described here, the actual selection of a base was based on the criteria indicated in the first paragraph of this section; the number of samples to be processed (4,096) was divided into the amount of time (14 milliseconds) allowed for processing. This gave a *total* memory cycle time of 3.42 microseconds for *all* accesses to each sample—assuming one sample per memory word. Because one and three sample-per-word (16 and 48 bit per word) memories led to less economical systems, the decision was made to go to two samples per word. This results in 6.84 microseconds *total* cycle time for *all* accesses to each sample. Based on this number, a curve of number of passes through the entire memory can be drawn as in Figure 6.

Assuming the previously stated minimum cycle of 1.5 microseconds is used, Figure 6 shows that the maximum number of passes through the entire memory is 4.56. In general, the use of fractional passes does not seem to offer any advantages except for two special cases:

1. Special operations at the start or end of a frame, such as time to operate on the address sequence control. Such operations probably have an effect of no greater than a few percent on the memory speed.

2. The use of a first processing step as new samples enter the memory. If a base of two is used, processing of samples could start just after* one-half of the samples have been loaded; if a base of four is used, processing could start just after three-fourths of the samples are loaded; for a base of B, processing could start just after $B - 1/B$ of the samples are loaded. Such a step was considered in this design, but was rejected because it did not result in either reducing the number of subsequent passes through the memory or in a significant simplification of the vector processor. (If a last pass through the vector processor were performed as data left the memory, the use of an input pass would have been profitable. The scheme, which was rejected,** would have used 4-½ passes through the memory. Processing would be base-2 at input and base-2 at output; three base-8 processing steps would occur between input and output; $2 \times 8^3 \times 2 = 2{,}048$ = number of samples per subband per frame.)

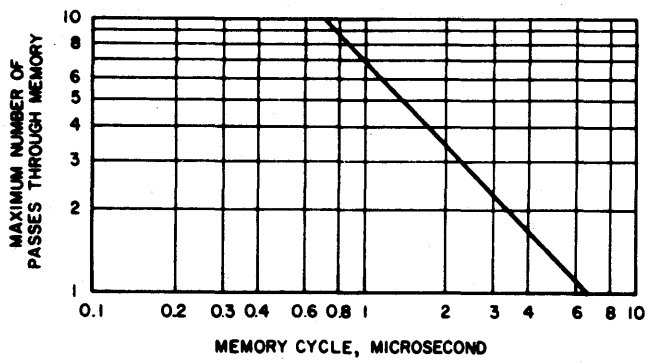


Figure 6—Maximum number of passes through memory for 6.84 microseconds for all accesses to each sample

Because the use of fractional passes was rejected, the selected memory was that for the next integral number of passes below the 4.6 (that is four) allowed by a 1.5-microsecond memory (see Figure 6). Since one of the four passes is required to load samples into the memory and simultaneously unload filters from the memory, this resulted in allowing three passes for processing. If the same base is used for all three steps with a total of 2,048 samples per subband, the base used must be $(2048)^{1/3} = 12.7$. A practical compromise is to use 16, 8 and 16 for the bases of the three passes. It is preferable that the passes be in the sequence listed since the symmetry (that is, having the first and last of the three passes use the same base) avoids unnecessary complexity in the address sequence control. Similarly for the 1,024 samples in the 7-millisecond case, bases of 16, 4 and 16 have been selected.

A base-8 processor* requires a total of five passes through the memory which must provide a cycle time of 1.4 microseconds and eliminates registers for eight words or 8×32 bits within the vector processor; this tradeoff is about even in dollars, but for a new design favors the slower memory in schedule risk and development cost. A base-4 processor** requires a memory cycle of 0.98 microsecond and eliminates registers for 12 words or 12×32 bits within the vector processor and might also save six 8-bit (scalar) multipliers, but requires much faster circuitry (arithmetic and otherwise) throughout the system. Thus no cost advantage appears to accrue to the lower base even though it requires a faster memory. Considering all such factors, the selected system appears optimum for its intended application since it is at worst no more expensive than the alternatives and is the most easily designed and developed.

* Here "just after" means when one more sample has been digitized.

** The use of a processing pass concomitant with data input appears to complicate the timing and does make the data flow more complex.

*Actually 8/8/4/8 or 8/4/8/8 (2,048 pulses per subband).

** Actually five passes of base-4 and one of base-2 (2,048 pulses per subband).

Burst processing

The concept of burst processing allows uninterrupted use of a period equivalent to five memory cycles for the operations within the vector processor. Thus it permits acceptably slow arithmetic circuits within the processor without a disproportionate increase in the number of buffer registers.

The read-write cycle associated with loading a new sample into the memory occurs once for each sample and occurs three times during processing of each sample; i.e., of each four memory cycles three are required for processing alone. Thus, while 16 memory cycles are used to read a new set of 16 operands per subband into the processor and transfer a set of 16 results per subband from the processor, the average time span during which these 16 memory cycles occur is actually $4/3 \times 16 = 21\text{-}\frac{1}{3}$ memory cycles.

On the other hand, as noted previously, if an internal base of two is used, processing in the base-16 vector processor cannot start until the ninth operand is available and cannot start until the thirteenth operand is available if an internal base of four is used. A more stringent limitation results from the decision to always exchange memory data; each time an operand is read from the memory, there must be a result ready to put back in the memory. If this decision is retained, two undesirable alternatives result: to perform all processing in one memory cycle using very fast circuits or to provide a redundant set of registers for 16 words. The latter would allow a procedure similar to that which is used with a random access memory large enough for two frames of data and in which one-half the storage is used for input–output while the other half is used for processing. Since neither alternative is pleasant, it is highly desirable that a method be conceived to increase the allowed processing time to four or five uninterrupted memory cycles.

The scheme adopted was to perform the transfer* of the 16 operands of the two subbands from the memory (through the vector multiplier) to the vector processor in one burst of 16 memory cycles. During the 16 cycles, new samples destined for the memory from the analog/digital converter are stored in an integrated circuit buffer of six words each of which represents one sample from each of the two subbands. The six-word buffer may store either five or six words, as illustrated in the timing diagram of Figure 2.

As shown in that figure, four pairs of samples are obtained during the first 16-memory-cycle burst used to communicate with the vector processor and a fifth pair is obtained soon thereafter. The five pairs are then transferred, in a burst, to the memory as an equal number of results are read out of the memory through the vector multiplier to the threshold circuits.

The procedure is repeated in bursts as depicted in the figure with every third input burst transferring six sample pairs rather than five.

SYSTEM DESCRIPTION

Address sequence control

The functions of the address sequence control (see Figure 1) are:

1. To determine the locations (addresses) in the random access memory (RAM) from which data is read and into which data is written.
2. To provide the sine/cosine table with a measure of the rotation angles (unit vectors) to be used to rotate vectors leaving the memory for the vector processor.

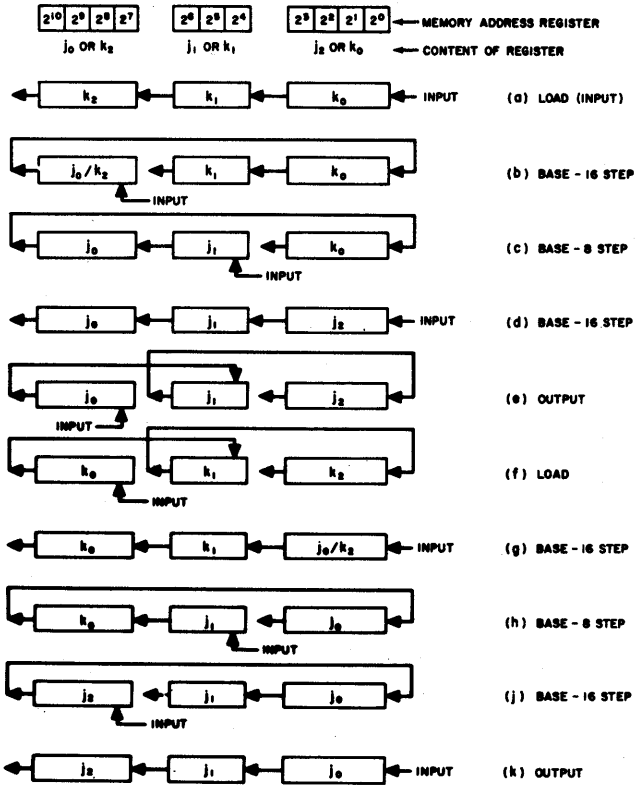


Figure 7—Address counter configurations for 2,048 samples per subband

* As each operand is transferred from memory to processor, a result (from a set of 16 previous operands) is transferred from the processor into the same memory position.

To avoid confusing the reader with a mathematical notion (of the fast Fourier transform procedure) involving many changeable subscripts, the explanation* of the address sequence control is in terms of exemplifying logic block diagrams for the 2,048 sample per subband case.

Description of processing sequence

Let

$$f(j) = \frac{1}{2048} \sum_{k=0}^{2047} m_k e^{i\theta jk} \tag{4}$$

where $\theta = 2\pi/2048$ and $i = \sqrt{-1}$. Let

$$j = j_2 128 + j_1 16 + j_0 \tag{5}$$

$$k = k_2 128 + k_1 16 + k_0 \tag{6}$$

where

$$j_2, j_0, k_2, k_0 = 0, 1, 2, \cdots 15 \tag{7}$$

$$j_1, k_1 = 0, 1, 2, \cdots 7 \tag{8}$$

$$f(j_2, j_1, j_0) = \frac{1}{2048} \sum_{k_0=0}^{15} \sum_{k_1=0}^{7} \sum_{k_2=0}^{15} m_{k_2 128 + k_1 16 + k_0}$$

$$\exp\left[i\theta(j_2 128 + j_1 16 + j_0)(k_2 128 + k_1 16 + k_0)\right]$$

$$= \frac{1}{16} \sum_{k_0=0}^{15} \exp\left[i\theta(j_2 128 + j_1 16 + j_0) k_0\right]$$

$$\sum_{k_1=0}^{7} \frac{1}{8} \exp\left[i\theta(j_1 16 + j_0) 16 k_1\right]$$

$$\sum_{k_2=0}^{15} \frac{1}{16} m_{k_2 128 + k_1 16 + k_0} \exp\left[i\theta j_0 128 k_2\right]$$

Step 1 (Base-16) Compute

$$f(j_0, k_1, k_0) = \frac{1}{16} \sum_{k_2=0}^{15} m_{k_2 128 + k_1 16 + k_0} \exp\left[i\theta j_0 128 k_2\right] \tag{9}$$

Step 2 (Base-8) Compute

$$f(j_0, j_1, k_0) = \frac{1}{8} \sum_{k_1=0}^{7} f(j_0, k_1, k_0) \exp\left[i\theta(j_1 16 + j_0) 16 k_1\right] \tag{10}$$

Step 3 (Base-16) Compute

$$f(j_0, j_1, j_2) = \frac{1}{16} \sum_{k_2=0}^{15} f(j_0, j_1, k_0)$$

$$\exp\left[i\theta(j_2 128 + j_1 16 + j_0) k_0\right] \tag{11}$$

* Also see appendix.

Address control for readout sequence

Equations (9), (10) and (11) constitute the three steps in the algorithm adopted for this processor and Equations (7) and (8) define the ranges of the j's and k's. The first step sorts the data according to the lowest digit of the frequency representation. Since the summation is over k_2, there are 128 different $f(j_0, k_1, k_0)$ for each j_0.

If a counter is organized in three sections corresponding to k_2, k_1 and k_0, in terms of the original input data, one counts in k_0 and carries from k_0 to k_1 and thence to k_2. This is shown in Figure 7a.

To form the set of $f(j_0, k_1, k_0)$ in Equation (9) (Step 1), the summation requires a count of k_2 and then progresses to cover all combinations of k_1 and k_0. It is accomplished by injecting the count pulse into the k_2 section of the counter and letting the carry flow to k_0 and thence to k_1. At the end of this step, the data has been sorted according to j_0 as shown in Figure 7(a).

The next two steps of the algorithm (Equations (10) and (11)) evaluate the sets $f(j_0, j_1, k_0)$ and $f(j_0, j_1, j_2)$ and proceed similarly with the counter inputs as shown in Figures 7(c) and (d), respectively. Due to special system requirements, the output must be read out in order of frequency. With the counter significance, in terms of j's shown in Figure 7(d), the carry path must be reconnected in order to read out in the proper frequency sequence. This is shown in Figure 7(e). Since this is by design* also the next input sequence, the connection for the output sequence in Figure 7(e) is the same for the next input sequence; thus Figure 7(f) is the same as Figure 7(e) with j's replaced by k's.

In Figure 7(b), k_2 carried to k_0 although carrying to either k_1 or k_0 was proper as long as all the combinations of k_1 and k_0 are eventually gone through. Similarly, in the step of Figure 7(g), k_2 carries to k_1 and thence to k_0 to again use the original counter configuration. As before, after the first step, the k_2 counter has the significance of j_0. The next two steps are again similarly performed as shown in Figures 7(h) and 7(j).

The output sequence shown in Figure 7(k) is identical to Figure 7(a), thus completing one cycle of counter configurations.

Memory data exchange cycling

The previous section explains the operation of the readout cycle and ignores the control cycle for returning processed data to the memory. The design described operates by exchanging a set of 16 (or eight) previ-

* To avoid a split-cycle memory or a faster memory.

ously processed data from the vector processor for a set of 16 (or eight) unprocessed data from the memory. This way, only 16 (or eight) memory cycles are required for each batch. Thus, the particular counter configuration of Figure 7 displaces the data after processing, by 16 memory positions. For example, after processing, the 16 pieces of data from $k_2 = 0, 1, 2, \ldots 15, k_1 = 0, k_0 = 0$ as shown in Figure 7(b), are returned to $k_2 = 0, 1, 2, \ldots 15, k_1 = 0, k_0 = 1$. Finally, the 16 pieces of data from $k_2 = 0, 1, 2, \ldots 15, k_1 = 7, k_0 = 15$ are returned to $k_2 = 0, 1, 2, \ldots 15, k_1 = 0, k_0 = 0$, which was the source of the first 16 pieces.

This last step requires an extra 16 (or eight) memory cycles in addition to the 2,048 cycles required for each pass of the Cooley–Tukey algorithm.

A simple concept to keep track of the memory address precession is through the description of index registers. Consider a memory with addresses enumerated 0, 1, 2, . . . k, . . . $2^{11} - 1$. This transfer may be hidden from outside world if an index adder is connected as shown in Figure 8.

Similarly a sequence of such transfers such as m_k to m_{k+p} followed by m_{k+p_1} to m_{k+p_2}, etc., can be disguised with an index register. For the purpose of the address sequence generator, the index register in Figure 8 can be made in the form of an accumulating register so that it always contains $\sum p_i$, modulo 2^{11}. Therefore, this configuration can keep track of any number of address shifts.

In Figure 7, if the counters shown are considered as equivalent (to the outside world) to the memory address register and if the modulo 2^{11} adder and index registers are interposed as shown in Figure 8, the corrections are made as follows:

	After Step	*Add to Index Register*
Phase 1	1(b)	2^0
	1(c)	2^7
	1(d)	2^4
Phase 2	1(g)	2^4
	1(h)	2^7
	1(j)	2^0

It is noted that once initialized the index register is never set to zero.

There are two alternate frames to be controlled. If the two occupy separate areas in the memory from addresses 0 to $2^{11} - 1$ and from 2^{11} to $2^{12} - 1$, the same address control may be used as identical operations delayed by one frame are performed in memory addresses that differ by 2^{11}. If we add 2^{11} to the index register for the second frame, the general operational area of the memory will be shifted to the other half. The simplest way to cope with the two alternate frames is to double the equipment for the address counters and index registers while sharing the index adder. In this way, the two sequences can be generated independently.

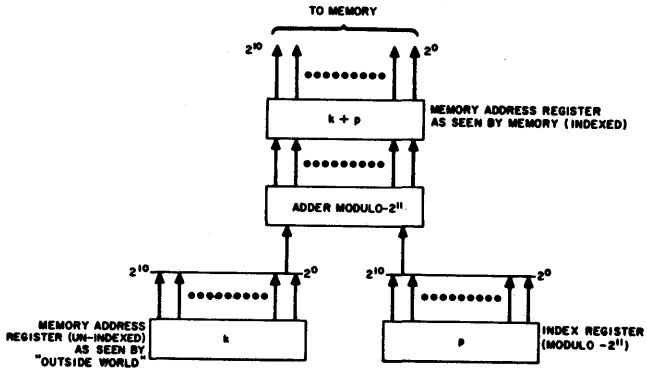


Figure 8—Memory address indexing

Phase shifter coefficients

The three processing steps Equations (9), (10) and (11) may be written as:

Step 1 (Base–16)

$$f(j_0, k_1, k_0) = \frac{1}{16} \sum_{k_2=0}^{15} m_{k_2128+k_116+k_0} \exp [ij_0(\pi/8)k_2] \quad (12)$$

Step 2 (Base–8)

$$f(j_0, j_1, k_0) = \frac{1}{8} \sum_{k_1=0}^{7} \{ f(j_0, k_1, k_0) \exp [ij_0(\pi/64)k_1] \} \exp [ij_1(\pi/4)k_1] \quad (13)$$

Step 3 (Base–16)

$$f(j_0, j_1, j_2) = \frac{1}{16} \sum_{k_0=0}^{15} \{ f(j_0, j_1, k_0) \exp [ij_0(\pi/1024)k_0] \exp [ij_1(\pi/64)k_0] \} \exp [ij_2(\pi/8)k_0] \quad (14)$$

These equations are implemented in a base-16 vector processor that provides vector rotation in multiples of $\pi/8$ (and hence $\pi/4$). This corresponds to the factored out exponent. The operations inside the big brackets in steps 2 and 3 are one operation to each "f" so that they may be performed as information transits from

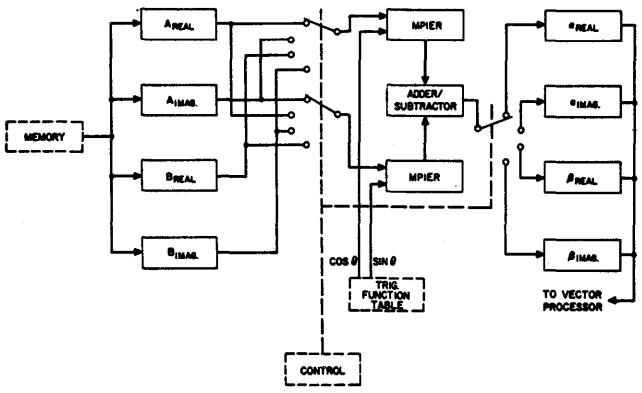


Figure 9—Vector multipler shown in phase rotation mode

the memory to the vector processor. The coefficients are functions of the address counter.

In Step 2, the rotations are in units of $\pi/64$. There are 128 possible entries in a table. The product of j_0k_1 can range from 0 to 105. If a 7-bit number is used for representing this product, the first two digits may be used to indicate the quadrant so that the table is reduced to 32 places. In Step 3, there are two exponents in the bracket. The multiples of $\pi/64$ can be treated the same as before except the multiplier is now j_1k_0. The other exponent involves multiples of $\pi/1024$. Since there are 226 different products of j_0k_0, a table of 226 places will be sufficient. The two vector multiplications may be performed in cascade.

Vector multiplier

The vector multiplier receives memory words at a rate of one each 1.71 microseconds and in the "phase rotation" mode (see Figure 9) receives the sine and cosine of the rotation angle at a similar rate from the trigonometric function table; in the "filter magnitude" mode (see Figure 10) used to obtain the squared magnitude no other input is required.

In both modes of operation, each 32-bit word coming from the memory is comprised of a 16-bit datum for each of the two subbands. The 16-bit quantity has an 8-bit real portion and an 8-bit imaginary portion. The real and imaginary parts of the A and B subband data are loaded (Figures 9 and 10) into the four 8-bit registers A_{real}, B_{real}, A_{imag} and B_{imag}. In the phase rotation mode (Figure 9) operation occurs in four sequences (one after another) to give (in the α and β registers):

1. $\alpha_{\text{real}} = \text{Real}\,[(A_{\text{real}} + iA_{\text{imag}})(\cos\theta + i\sin\theta)]$
$= A_{\text{real}}\cos\theta - A_{\text{imag}}\sin\theta$
2. $\alpha_{\text{imag}} = \text{Imag}\,[(A_{\text{real}} + iA_{\text{imag}})(\cos\theta + i\sin\theta)]$
$= A_{\text{real}}\sin\theta + A_{\text{imag}}\cos\theta$
3. $\beta_{\text{real}} = B_{\text{real}}\cos\theta - B_{\text{imag}}\sin\theta$
4. $\beta_{\text{imag}} = B_{\text{real}}\sin\theta + B_{\text{imag}}\cos\theta$

Within each sequence, the two required multiplications occur simultaneously in the two (scalar) multipliers (Mpier) and the products are immediately added in the adder/subtractor. A complete memory cycle, less only setting time for the A and B registors, is available for four sequences. An allowance of 400 nanoseconds for each sequence seems realistic.

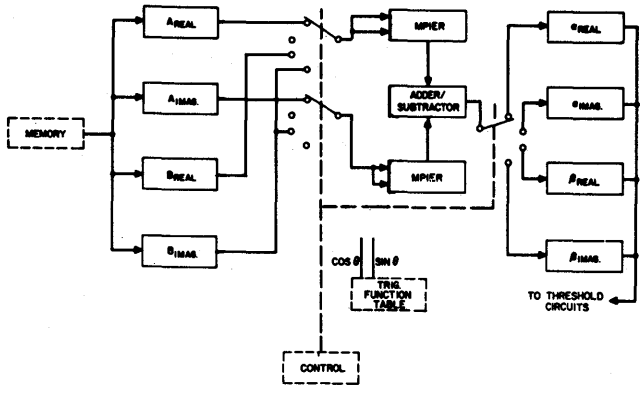


Figure 10—Vector multiplier shown in filter magnitude mode

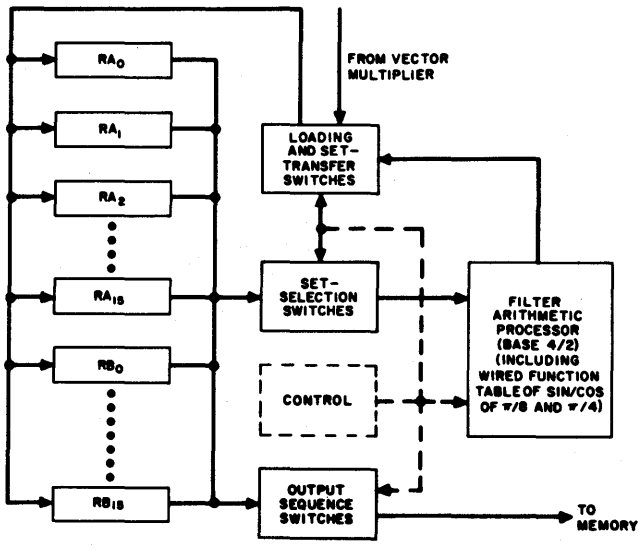


Figure 11—Vector processor

TABLE II—Operation Sequences Within Filter Arithmetic Processor

	Base-4	Base-8	Base-16
First Step	$R_0 = O_0 + O_1 + O_2 + O_3$ $R_1 = O_0 - O_1 + O_2 - O_3$ $R_2 = O_0 + iO_1 - O_2 - iO_3$ $R_3 = O_0 - iO_1 - O_2 + iO_3$	$R_0 = O_0 + O_2 + O_4 + O_6$ $R_2 = O_0 - O_2 + O_4 - O_6$ $R_4 = O_0 + iO_2 - O_4 - iO_6$ $R_6 = O_0 - iO_2 - O_4 + iO_6$	$R_0 = O_0 + O_4 + O_8 + O_{12}$ $R_4 = O_0 - O_4 + O_8 - O_{12}$ $R_8 = O_0 + iO_4 - O_8 - iO_{12}$ $R_{12} = O_0 - iO_4 - O_8 + iO_{12}$
	$R_4 = O_4 + O_5 + O_6 + O_7$ $R_5 = O_4 - O_5 + O_6 - O_7$ $R_6 = O_4 + iO_5 - O_6 - iO_7$ $R_7 = O_4 - iO_5 - O_6 + iO_7$	$R_1 = O_1 + O_3 + O_5 + O_7$ $R_3 = O_1 - O_3 + O_5 - O_7$ $R_5 = O_1 + iO_3 - O_5 - iO_7$ $R_7 = O_1 - iO_3 - O_5 + iO_7$	$R_1 = O_1 + O_5 + O_9 + O_{13}$ $R_5 = O_1 - O_5 + O_9 - O_{13}$ $R_9 = O_1 + iO_5 - O_9 - iO_{13}$ $R_{13} = O_1 - iO_5 - O_9 + iO_{13}$
	$R_8 = O_8 + O_9 + O_{10} + O_{11}$ $R_9 = O_8 - O_9 + O_{10} - O_{11}$ $R_{10} = O_8 + iO_9 - O_{10} - iO_{11}$ $R_{11} = O_8 - iO_9 - O_{10} + iO_{11}$	$R_8 = O_8 + O_{10} + O_{12} + O_{14}$ $R_{10} = O_8 - O_{10} + O_{12} - O_{14}$ $R_{12} = O_8 + iO_{10} - O_{12} - iO_{14}$ $R_{14} = O_8 - iO_{10} - O_{12} + iO_{14}$	$R_2 = O_2 + O_6 + O_{10} + O_{14}$ $R_6 = O_2 - O_6 + O_{10} - O_{14}$ $R_{10} = O_2 + iO_6 - O_{10} - iO_{14}$ $R_{14} = O_2 - iO_6 - O_{10} + iO_{14}$
	$R_{12} = O_{12} + O_{13} + O_{14} + O_{15}$ $R_{13} = O_{12} - O_{13} + O_{14} - O_{15}$ $R_{14} = O_{12} + iO_{13} - O_{14} - iO_{15}$ $R_{15} = O_{12} - iO_{13} - O_{14} + iO_{15}$	$R_9 = O_9 + O_{11} + O_{13} + O_{15}$ $R_{11} = O_9 - O_{11} + O_{13} - O_{15}$ $R_{13} = O_9 + iO_{11} - O_{13} - iO_{15}$ $R_{15} = O_9 - iO_{11} - O_{13} + iO_{15}$	$R_3 = O_3 + O_7 + O_{11} + O_{15}$ $R_7 = O_3 - O_7 + O_{11} - O_{15}$ $R_{11} = O_3 + iO_7 - O_{11} - iO_{15}$ $R_{15} = O_3 - iO_7 - O_{11} + iO_{15}$
Second Step	No second step	$R_0 = O_0 + O_1$ $R_1 = O_0 - O_1$ $R_2 = O_2 + iO_3$ $R_3 = O_2 - iO_3$	$R_0 = O_0 + O_1 + O_2 + O_3$ $R_1 = O_0 - O_1 + O_2 - O_3$ $R_2 = O_0 + iO_1 - O_2 - iO_3$ $R_3 = O_0 - iO_1 - O_2 + iO_3$
		$R_4 = O_4 + e^{i\pi/4} O_5$ $R_5 = O_4 - e^{i\pi/4} O_5$ $R_6 = O_6 + ie^{i\pi/4} O_7$ $R_7 = O_6 - ie^{i\pi/4} O_7$	$R_4 = O_4 + e^{i\pi/4} O_5 + iO_6 + ie^{i\pi/4} O_7$ $R_5 = O_4 - e^{i\pi/4} O_5 + iO_6 - ie^{i\pi/4} O_7$ $R_6 = O_4 + ie^{i\pi/4} O_5 - O_6 + e^{i\pi/4} O_7$ $R_7 = O_4 - ie^{i\pi/4} O_5 - O_6 - e^{i\pi/4} O_7$
		$R_8 = O_8 + O_9$ $R_9 = O_8 - O_9$ $R_{10} = O_{10} + iO_{11}$ $R_{11} = O_{10} - iO_{11}$	$R_8 = O_8 + e^{i\pi/8} O_9 + e^{i\pi/4} O_{10} + e^{i3\pi/8} O_{11}$ $R_9 = O_8 - e^{i\pi/8} O_9 + e^{i\pi/4} O_{10} - e^{ie\pi/8} O_{11}$ $R_{10} = O_8 + ie^{i\pi/8} O_9 - e^{i\pi/4} O_{10} - ie^{i3\pi/8} O_{11}$ $R_{11} = O_8 - ie^{i\pi/8} O_9 - e^{i\pi/4} O_{10} + ie^{i3\pi/8} O_{11}$
		$R_{12} = O_{12} + e^{i\pi/4} O_{13}$ $R_{13} = O_{12} - e^{i\pi/4} O_{13}$ $R_{14} = O_{14} + ie^{i\pi/4} O_{15}$ $R_{15} = O_{14} - ie^{i\pi/4} O_{15}$	$R_{12} = O_{12} + e^{i3\pi/8} O_{13} + ie^{i\pi/4} O_{14} - e^{i\pi/8} O_{15}$ $R_{13} = O_{12} - e^{i3\pi/8} O_{13} + ie^{i\pi/4} O_{14} + e^{i\pi/8} O_{15}$ $R_{14} = O_{12} + ie^{3\pi/8} O_{13} - ie^{i\pi/4} O_{14} + ie^{i\pi/8} O_{15}$ $R_{15} = O_{12} - ie^{3\pi/8} O_{13} - ie^{i\pi/4} O_{14} - ie^{i\pi/8} O_{15}$

NOTES:

1) R = Result; O = Operand
2) $e^{i\theta} = \cos\theta + i \sin\theta$
3) $e^{i(\pi/2 - \theta)} = \sin\theta + i \cos\theta$

Operation of the vector multiplier in the filter magnitude mode is essentially similar (Figure 10) except that only two sequences are performed. These result in the following α and β register contents:

1. $\alpha_{\text{real}} = A_{\text{real}}^2 + A_{\text{imag}}^2$
2. $\beta_{\text{real}} = B_{\text{real}}^2 + B_{\text{imag}}^2$

Vector processor

Within the vector processor, fast Fourier processing employs either a base-4 or base-2 procedure. For an (externally viewed) overall base of 16, two base-4 steps occur and for a base-8 procedure, a base-4 and base-2 step are combined.

Conceptually and equipment-wise, the vector processor (Figure 11) can be viewed as two 16-register sets of 16-bit registers R_{A0} to R_{A15} and R_{B0} to R_{B15}. Each set stores the 16 data associated with the processing of a one-subband group of 16 operands; thus the 16 16-bit registers can contain one *G*-16 (see Summary System Description) or two *G*-8's or four *G*-4's. (Only the *G*-16 case is described.)

As a new *G*-16 enters the vector processor from the vector multiplier, it is appropriately loaded into the 32

TABLE III—Rewritten Operation Sequences Within Filter Arithmetic Processor

	Base-4	Base-8	Base-16
First Step	$R_0 = (0_0 + 0_2) + (0_1 + 0_3)$ $R_1 = (0_0 + 0_2) - (0_1 + 0_3)$ $R_2 = (0_0 - 0_2) + i(0_1 - 0_3)$ $R_3 = (0_0 - 0_2) - i(0_1 - 0_3)$	$R_0 = (0_0 + 0_4) + (0_2 + 0_6)$ $R_2 = (0_0 + 0_4) - (0_2 + 0_6)$ $R_4 = (0_0 - 0_4) + i(0_2 - 0_6)$ $R_6 = (0_0 - 0_4) - i(0_2 - 0_0)$	$R_0 = (0_0 + 0_8) + (0_4 + 0_{12})$ $R_4 = (0_0 + 0_8) - (0_4 + 0_{12})$ $R_8 = (0_0 - 0_8) + i(0_4 - 0_{12})$ $R_{12} = (0_0 - 0_8) - i(0_4 - 0_{12})$
	$R_4 = (0_4 + 0_6) + (0_5 + 0_7)$ $R_5 = (0_4 + 0_6) - (0_5 + 0_7)$ $R_6 = (0_4 - 0_6) + i(0_5 - 0_7)$ $R_7 = (0_4 - 0_6) - i(0_5 - 0_7)$	$R_1 = (0_1 + 0_5) + (0_3 + 0_7)$ $R_3 = (0_1 + 0_5) - (0_3 + 0_7)$ $R_5 = (0_1 - 0_5) + i(0_3 - 0_7)$ $R_7 = (0_1 - 0_5) - i(0_3 - 0_7)$	$R_1 = (0_1 + 0_9) + (0_5 + 0_{13})$ $R_5 = (0_1 + 0_9) - (0_5 + 0_{13})$ $R_9 = (0_1 - 0_9) + i(0_5 - 0_{13})$ $R_{13} = (0_1 - 0_9) - i(0_5 - 0_{13})$
	$R_8 = (0_8 + 0_{10}) + (0_9 + 0_{11})$ $R_9 = (0_8 + 0_{10}) - (0_9 + 0_{11})$ $R_{10} = (0_8 - 0_{10}) + i(0_9 - 0_{11})$ $R_{11} = (0_8 - 0_{10}) - i(0_9 - 0_{11})$	$R_8 = (0_8 + 0_{12}) + (0_{10} + 0_{14})$ $R_{10} = (0_8 + 0_{12}) - (0_{10} + 0_{14})$ $R_{12} = (0_8 - 0_{12}) + i(0_{10} - 0_{14})$ $R_{14} = (0_8 - 0_{12}) - i(0_{10} - 0_{14})$	$R_2 = (0_2 + 0_{10}) + (0_6 + 0_{14})$ $R_6 = (0_2 + 0_{10}) - (0_6 + 0_{14})$ $R_{10} = (0_2 - 0_{10}) + i(0_6 - 0_{14})$ $R_{14} = (0_2 - 0_{10}) - i(0_6 - 0_{14})$
	$R_{12} = (0_{12} + 0_{14}) + (0_{13} + 0_{15})$ $R_{13} = (0_{12} + 0_{14}) - (0_{13} + 0_{15})$ $R_{14} = (0_{12} - 0_{14}) + i(0_{13} - 0_{15})$ $R_{15} = (0_{12} - 0_{14}) - i(0_{13} - 0_{15})$	$R_9 = (0_9 + 0_{13}) + (0_{11} + 0_{15})$ $R_{11} = (0_9 + 0_{13}) - (0_{11} + 0_{15})$ $R_{13} = (0_9 - 0_{13}) + i(0_{11} - 0_{15})$ $R_{15} = (0_9 - 0_{13}) - i(0_{11} - 0_{15})$	$R_3 = (0_3 + 0_{11}) + (0_7 + 0_{15})$ $R_7 = (0_3 + 0_{11}) - (0_7 + 0_{15})$ $R_{11} = (0_3 - 0_{11}) + i(0_7 - 0_{15})$ $R_{15} = (0_3 - 0_{11}) - i(0_7 - 0_{15})$
Second Step	No second step	$R_0 = 0_0 + 0_1$ $R_1 = 0_0 - 0_1$ $R_2 = 0_2 + i0_3$ $R_3 = 0_2 - i0_3$	$R_0 = (0_0 + 0_2) + (0_1 + 0_3)$ $R_1 = (0_0 + 0_2) - (0_1 + 0_3)$ $R_2 = (0_0 - 0_2) + i(0_1 - 0_3)$ $R_3 = (0_0 - 0_2) - i(0_1 - 0_3)$
		$R_4 = 0_4 + (1 + i)\, 0_5 \sin \pi/4$ $R_5 = 0_4 - (1 + i)\, 0_5 \sin \pi/4$ $R_6 = 0_6 + (1 + i)\, 0_7 \sin \pi/4$ $R_7 = 0_6 - (1 + i)\, 0_7 \sin \pi/4$	$R_4 = (0_4 + i0_6) + (0_5 + i0_7)(1 + i) \sin \pi/4$ $R_5 = (0_4 + i0_6) - (0_5 + i0_7)(1 + i) \sin \pi/4$ $R_6 = (0_4 - i0_6) + (i0_5 + 0_7)(1 + i) \sin \pi/4$ $R_7 = (0_4 - i0_6) - (i0_5 + 0_7)(1 + i) \sin \pi/4$
		$R_8 = 0_8 + 0_9$ $R_9 = 0_8 - 0_9$ $R_{10} = 0_{10} + i0_{11}$ $R_{11} = 0_{10} - i0_{11}$	$R_8 = 0_8 + (1 + i)\, 0_{10} \sin \pi/4 + (0_9 + i0_{11}) \cos \pi/8 + (i0_9 + 0_{11}) \sin \pi/8$ $R_9 = 0_8 + (1 + i)\, 0_{10} \sin \pi/4 - (0_9 + i0_{11}) \cos \pi/8 - (i0_9 + 0_{11}) \sin \pi/8$ $R_{10} = 0_8 - (1 + i)\, 0_{10} \sin \pi/4 + (i0_9 + 0_{11}) \cos \pi/8 - (0_9 + i0_{11}) \sin \pi/8$ $R_{11} = 0_8 - (1 + i)\, 0_{10} \sin \pi/4 - (i0_9 + 0_{11}) \cos \pi/8 + (0_9 + i0_{11}) \sin \pi/8$
		$R_{12} = 0_{12} + (1 + i)\, 0_{13} \sin \pi/4$ $R_{13} = 0_{12} - (1 + i)\, 0_{13} \sin \pi/4$ $R_{14} = 0_{14} + (1 + i)\, 0_{15} \sin \pi/4$ $R_{15} = 0_{14} - (1 + i)\, 0_{15} \sin \pi/4$	$R_{12} = 0_{12} - (1 - i)\, 0_{14} \sin \pi/4 + (i0_{13} - 0_{15}) \cos \pi/8 + (0_{13} - i0_{15}) \sin \pi/8$ $R_{13} = 0_{12} - (1 - i)\, 0_{14} \sin \pi/4 - (i0_{13} - 0_{15}) \cos \pi/8 - (0_{13} - i0_{15}) \sin \pi/8$ $R_{14} = 0_{12} + (1 - i)\, 0_{14} \sin \pi/4 - (0_{13} - i0_{15}) \cos \pi/8 + (i0_{13} - 0_{15}) \sin \pi/8$ $R_{15} = 0_{12} + (1 - i)\, 0_{14} \sin \pi/4 + (0_{13} - i0_{15}) \cos \pi/8 - (i0_{13} - 0_{15}) \sin \pi/8$

NOTES:

1) R = Result; 0 = Operand
2) $e^{i\theta} = \cos \theta + i \sin \theta$
3) $e^{i(\pi/2 - \theta)} = \sin \theta + i \cos \theta$
4) Since $\cos \pi/4 = \sin \pi/4$, $e^{i\pi/4} = (1 + i) \sin \pi/4$
5) $0 = 0_{Real} + i0_{Imag}$ and $(1 + i)\, 0 = (0_{Real} - 0_{Imag}) + i(0_{Real} + 0_{Imag})$

registers by the loading and set-transfer switches.* At the same time the old G-16 (the result of the previous operation) is transferred out of the registers and into the random access memory where it replaces the G-16 being read in. Transfer from the memory occurs under control of the output sequence switches. Thus, during a 16 memory cycle burst, the processed (old) G-16 in the vector processor is exchanged for a (new) G-16 from the memory; as it passes through the vector multiplier, the data of the new G-16 are multiplied by the proper unit vectors.

During the subsequent five memory cycles,** the actual processing of the G-16 occurs. In this process a sequence of four-register-sets is selected by the set selection switches, transferred to the filter arithmetic processor (FAP) where they are subjected to a base-4 fast Fourier procedure and then returned to the four registers from which they came. In order to process the G-16, a total of four sets must "go through" the FAP per base-4 step per subband. This is illustrated by the base-16 column at the right of Table II. In that table, it is seen that the first base-4 step uses (for each subband) operands O_0, O_4, O_8 and O_{12} to give results R_0, R_4, R_8 and R_{12} and uses operands O_1, O_5, O_9 and O_{13} to give results R_1, R_5, R_9 and R_{13}, and so on. The second step is described by the lower half of the base-16 column of Table II.

Table II presents the fast Fourier arithmetic in its most conventional form. The actual equations chosen to be mechanized are shown in Table III in which the equations have been rewritten to reduce the number of multiplier circuits required. The FAP itself is shown in Figure 12.

As demonstrated by Table III, the operations performed on the (input vector) operands and on combinations of the operands consist of multiplication by ± 1 and $\pm i$ $(= \sqrt{-1})$, addition and subtraction and multiplication by the sine of $\pi/4$ and/or the sine and cosine of $\pi/8$. All operations except multiplication by

* The switches are a part of the control and are considered separately only to facilitate the description.

** When six memory cycles are available for processing, the sixth cycle is a "dead period."

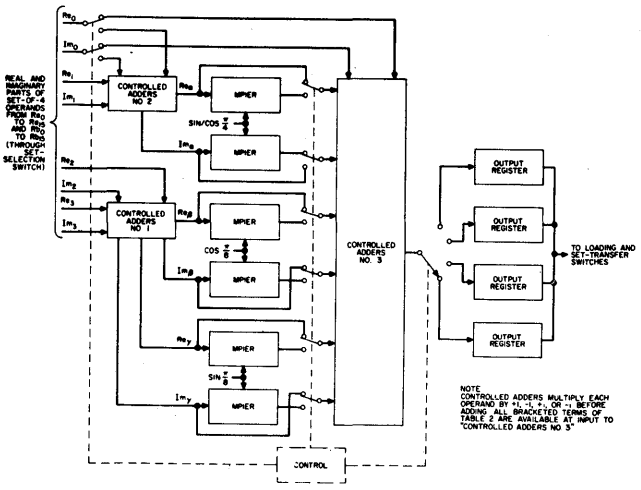


Figure 12—Filter arithmetic processor

the sines and the cosine occur in the "controlled adders" of Figure 12. In the controlled adders, pairs of vectors are added to or subtracted from each other either directly or after prior multiplication by ± 1 or $\pm i$. The results of the addition or subtraction may also be multiplied by ± 1 or $\pm i$.

Multiplication of the appropriate information by functions of $\pi/4$ and $\pi/8$ is performed in the six 8-bit (scalar) multipliers shown in Figure 12; these multipliers are similar to those used in the vector multiplier.

Since each word entering the vector processor contains one operand per subband (and there are two subbands) and since four sets of four operands are processed in the vector processor for each of the two base-4 steps which comprise a base-16 step, a total of $2 \times 4 \times 2 = 16$ sets of four operands must be processed by the FAP in 16 substeps during each five memory cycle period of 8.57 microseconds. This gives time of 0.53 microsecond per substep. Note that the period of 0.53 microsecond is not the time from the insertion of operands (into the input set of controlled adders) until the availability of the results in the four output registers. (As long as the delay is not great enough to endanger *system* timing the allowable time is a function of the overall vector processor design and system timing is not a problem because the first four sets of results for both subbands can be made available to the memory at the end of only 10 of the 16 substeps.) The significance of the 0.53 microsecond is that no procedure in the FAP may take a time approaching 0.53 microsecond; the longest procedure in the FAP is the 8-bit scalar multiplication and this requires only 0.3 microsecond. Thus, there are no special problems in the mechanization of the vector processor.

Sine/cosine table

At a rate of one set per 1.71-microsecond memory cycle, the sine/cosine table must provide sines and cosines of the multiples of the angle $2\pi/2048$; that is, of the angles $\theta(2\pi/2048)$ where θ is an integer between 0 and 2047. The angles are specified to the sine/cosine table by the j's and k's generated by the address sequence control. An internal function of the sine/cosine table is to compute (from the j's and k's) the value of θ which is to be used as the independent variable in entering the table. In Figure 13, which is the block diagram of the sine/cosine table, the value of θ is computed by the multiplier, Mpier. (This operation is trivial and is not described here.) Output of the multiplier is an 11-bit value of θ.

The value of θ is converted to a 9-bit positive number θ_T which is ultimately used to obtain the appropriate sine and cosine of an angle between 0 and $\pi/4$; that is, all sine and cosine values are computed for angles no greater than $\pi/4$ and the computed first-octant sine and cosine are used according to the trigonometric identities for complementary and supplementary angles and for angles which differ by π radians. These identities translate into the algorithms of Table IV which demonstrates how the angle θ is translated to the first octant angle θ_T or its two's complement and used in computing the sine and cosine.

Computation of the sine and cosine of θ_T is performed using a five-point table and linear interpolation between the five points 0, $\pi/16$, $\pi/8$, $3\pi/16$ and $\pi/4$. The inaccuracy of the interpolation is less than one part in 200 over the range of interest. Of greater import is the fact that the magnitude of the error is less than 2^{-8}

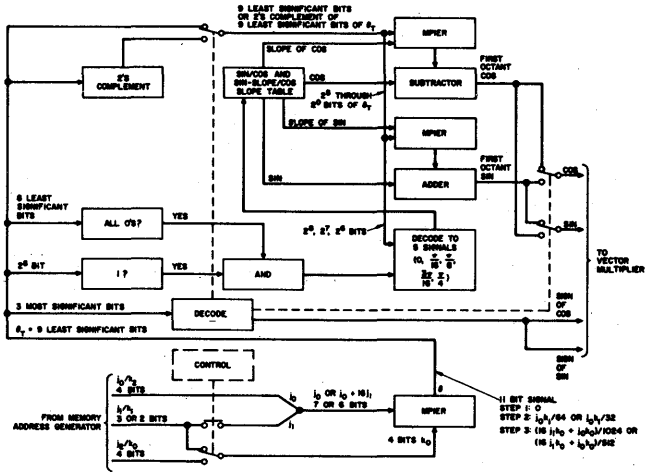
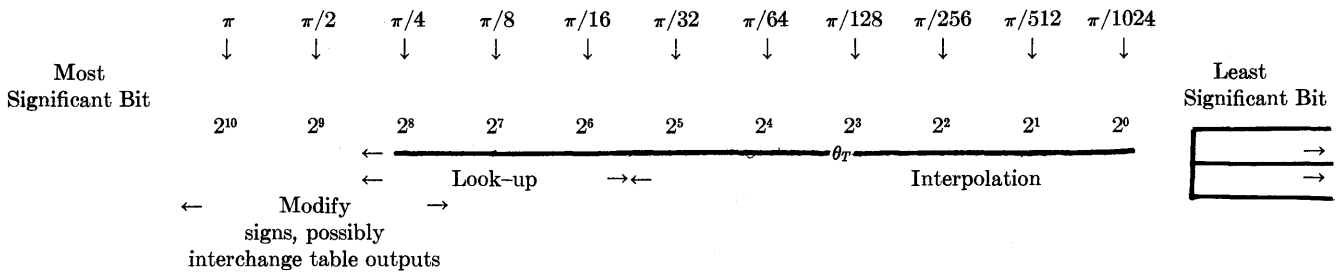


Figure 13—Sine/cosine table block diagram

TABLE IV—Determination of First-Octant Angle Used in Sine/Cosine Computation and Related Rules



If	Compute sine/cosine as follows
1. $\theta_T \leq 2^8$	Use θ_T.
2. $2^8 < \theta_T \leq 2^9$	Use 2's complement of θ_T. Interchange sine and cosine outputs.
3. $2^9 < \theta_T \leq (2^9 + 2^8)$	Use θ_T. Put − sign on cosine and then interchange sine and cosine.
4. $(2^9 + 2^8) < \theta_T \leq 2^{10}$	Use 2's complement of θ_T. Put − sign on sine.
5. $2^{10} < \theta_T \leq \theta_{max} = 2^{11} - 1$	Using all bits except 2^{10} do as above (1 to 4) and reverse all signs at output.

while the eight arithmetic bits employed represent sign and seven magnitude bits; thus the maximum error is less than one-half of the least significant bit.

The computational procedure employs a wired table of the sines and cosines of the selected five points and also supplies the slope of the sine and cosine at the first four points. (Since $\theta_T \leq \pi/4$, no slope is required beyond $\pi/4$.) Each slope is provided to one of the two multipliers which also receive the six least significant bits of θ_T; these represent the difference between θ_T and the next lower argument of the tabulated values of sine and cosine. Thus, the products of the two multipliers represent the amounts to be added to the tabular value of the sine and subtracted from the tabular value of the cosine* to obtain the function of θ_T. Addition and subtraction occur in the adder and subtractor as shown in Figure 13 and selection of sine and cosine and assignment of the proper signs are performed according to Table IV.

SUMMARY

A fast Fourier digital processor designed for the real-time filtering of radar signals has been described. This processor design can perform direct and inverse transforms and thus is also applicable to communications systems, correlation and pattern matching, convolution and other processes using the discrete Fourier transform. The design could also be used for a stand-alone Fourier transform system or for a system to be used in conjunction with a general purpose computer.

In the real-time radar application, the processor replaces 512 (analog) filters of 143 Hz bandwidth or 1024 filters of 72 Hz bandwidth.

The processor is employed on a full time basis to perform the Fourier transform. Thus, it can be designed to do this task more efficiently and at lower cost than would a general purpose computer. Design concepts which make this possible are:

1. The use of a fast Fourier base of 16 (rather than two) in order to reduce the memory and logic circuit requirements,
2. The use of a timing scheme (burst processing) which maximizes the uninterrupted periods allowed for processing,
3. Simple flow of information through and within the processor,
4. Separation of functions to be performed from one another and performing these functions in special purpose functional equipment.

* The slope of the cosine curve is negative.

BIBLIOGRAPHY

1 M I SKONIK
Introduction to radar systems
pp 113–126; 132–137; 151–162; 408–418
McGraw-Hill Book Company Inc New York 1962

2 W W MAQUIRE
Application of pulsed doppler to airborne radar systems
Proc of the National Conference on Aeronautical Electronics pp 291–295 Dayton Ohio 1958

3 G D BERGLAND
A guided tour of the fast fourier transform
IEEE Spectrum Vol 6 No 7 pp 41–52 July 1969

4 W T COCHRAN et al
What is the fast fourier transform?
IEEE Transactions on Audio and Electroacoustics Vol AU-15 No 2 pp 45–55 June 1967
5 B J THOMPSON B J GOLDSTONE
Digital signal processing
Electronic Progress Vol 12 No 1 pp 5–11 1968 and Computer Design Vol 8 No 5 pp 44–49 May 1969
6 D J POVEJSIL R S RAVEN P WATERMAN
Airborne radar
Boston Technical Publishers pp 311–331 Boston 1961
7 L N RIDENOUR EDITOR
Radar systems engineering
McGraw-Hill Book Company Inc pp 3–6, 123–126 New York 1947
8 J W COOLEY J W TUKEY
An algorithm for the machine calculation of complex fourier series
Mathematics of Computation Vol 19 No 2 pp 297–301 Aprill 1965
9 G D BERGLAND
A fast fourier transform algorithm using base 8 iterations
Mathematics of Computation Vol 22 No 2 pp 275–279 April 1968
10 W M GENTLEMAN G SANDE
Fast fourier transforms—for fun and profit
Proc Fall Joint Computer Conference Vol 29 pp 563–578 1966
11 J W COOLEY
Applications of the fast fourier transform method
Address given at the IBM Scientific Computing Symposium Digital Simulation of Continuous Systems Thomas J Watson Research Center Yorktown Heights June 20–22 1966
12 *IEEE transactions on audio and electroacoustics*
Special Issue on Fast Fourier Transform Vol AU-15 No 2 June 1967
13 *IEEE transactions on audio and electroacoustics*
Special Issue on Fast Fourier Transform Vol AU-17 No 2 June 1969
14 *IEEE transactions on audio and electroacoustics*
Special Issue on Digital Filters Vol AU-16 No 3 September 1968
15 R MCCULLOUGH
A real-time digital spectrum analyzer
Stanford Electronics Laboratories Scientific Report No 23 SU-SEL-099 Standford University Stanford California November 1967 and PhD Thesis Stanford University Stanford California December 1967
16 R R SHIVELY
A digital processor to generate spectra in real time
IEEE Transactions on Computers Vol C-17 No 5 pp 485–491 May 1968
17 C WEINSTEIN
Roundoff noise in floating point fast fourier transform
IEEE Transactions on Audio and Electroacoustics Vol AU-17 No 3 pp 209–215 September 1969
18 P D WELCH
A fixed point fast fourier transform error analysis
IEEE Transactions on Audio and Electroacoustics Vol AU-17 No 2 pp 151–157 June 1969
19 W R BENNETT
Spectra of quantized signals
Bell System Technical Journal Vol 27 No 3 pp 446–472 July 1948

APPENDIX

Fast Fourier transform digital processing and digital filters

During the last two decades, the problem of extracting or filtering out a small signal from considerably more powerful noise has been given much practical and theoretical attention for real-time processes such as radar and sonar signal processing and for delayed-time processes such as telemetry data reduction. In general, the filters and the real-time processors have been limited in complexity and conceptual sophistication by the requirement that they be realizable with acceptable quantities of hardware.

The delayed-time processes could often use very large computers, but were limited by the long, slow iterative procedures. As a result, significant effort was devoted to computational procedures. Probably the most useful of these has been the application of Fourier methods. However, as larger and larger complexes of data were attacked, these methods became more and more unwieldy because the amount of computation rose approximately as the square of the amount of data involved. Thus, if data were collected at fixed intervals over one "frame" (during which time N samples are collected), the computational effort involved in processing would be proportional to the square of the duration of the frame or for N measurements would vary directly with N^2. This is true since Fourier spectrum analysis, in digital form, with frequency resolution matched to the length of time the signal is under observation, is of the form

$$F_j = \sum_{k=0}^{N-1} A_k e^{ijk(2\pi/N)} \qquad \text{(A-1)}$$

where

A_k = the kth sampled values of the signal, $k = 0, 1, 2, \ldots, N-1$

$i = \sqrt{-1}$ and both A's and F's are complex (in-phase and quadrature sampled)

Brute force calculations required N^2 operations since the NF_j's are to be evaluated and each is the sum of N products. In high PRF processing, N is typically 1,000 or larger so that the amount of arithmetic by this method is prohibitively large for real-time operation.

An algorithm developed by J. W. Cooley and J. W. Tukey[8] reduces the computational load to $N \log_B N$ where B is the base (typically a power of 2 such as 2, 4, 8 or 16) to which the logarithm of N was taken and also represents the number of data from the full set of N which are processed in each substep of the procedure.

TABLE A-1—Calculating Time

N Number of Points	Time (in minutes) Conventional Method	Fast Method
512	0.17	0.0
1,024	0.67	0.0
2,048	2.70	0.01
4,096		0.03
8,192		0.07
16,384		0.16

Table A-1 provides a comparison of the time for calculating a Fourier transform by a popular conventional method and by the faster Cooley-Tukey method. These are proportional to N^2 and $N \log_2 N$, respectively. Both methods were programmed in basic FORTRAN and run on the IBM 7094.*

This radical improvement makes Fourier methods directly applicable to real-time all-digital processing of sensor information with presently available computer memories and integrated circuits. The Cooley-Tukey method will be described step-by-step, in terms of the procedures involved but without any attempt at proving their equivalence to the direct method. For proof of the equivalence of the two methods, the reader is referred to the references.

For the radar application, the important filter characteristic is the so-called amplitude frequency response, or simply frequency response, $H(j\omega)$ which is independent of the phase of the signal at the input of the filters. This may be analyzed by either Fourier transforms or by Fourier series.

The calculation of the Fourier series of Equation (A-1) can be performed digitally by calculating the complex discrete Fourier series in the form:

$$F = \sum_{k=0}^{K-1} W_k A_k \qquad \text{(A-2)}$$

where

A_k = the kth input sample pair (the real part is the in-phase component and the imaginary part is the quadrature component)

W_k = the kth complex member of the weighting sequence

F = the complex filter response which was determined by the weighting sequence $W_0, W_1, \ldots, W_{K-1}$

* From reference 12. Note that the IBM 7094 computer is slow compared to present generation airborne computers.

TABLE A-2—First Step of Fast Fourier or Cooley-Tukey Process

(16 samples per frame: B = Base = 2, S = Samples, T = Result of first step)

$S_1 + S_9 = T_1$	
$S_2 + S_{10} = T_2$	$e^{i0} = 1 + i0 = 1$
$S_3 + S_{11} = T_3$	$e^{i\pi} = -1 + i0 = -1$
$S_4 + S_{12} = T_4$	
$S_5 + S_{13} = T_5$	
$S_6 + S_{14} = T_6$	
$S_7 + S_{15} = T_7$	
$S_8 + S_{16} = T_8$	
$S_1 - S_9 = T_9$	
$S_2 - S_{10} = T_{10}$	
$S_3 - S_{11} = T_{11}$	
$S_4 - S_{12} = T_{12}$	
$S_5 - S_{13} = T_{13}$	
$S_6 - S_{14} = T_{14}$	
$S_7 - S_{15} = T_{15}$	
$S_8 - S_{16} = T_{16}$	

Here, the magnitudes of the sequence of W's determine the filter shape while the phase angles of the W's determine the frequencies of maximum filter response. Thus, we may form a bank of N filters using equation (A-2) and the arithmetic requirement can be greatly reduced by the use of the fast transform algorithm if the following reasonable restrictions are accepted.

1. All filters in the bank are the same shape.
2. Filter response frequencies for different filters are uniformly spaced with a spacing of

$$\frac{\text{sampling frequency}}{\text{any power of 2}}.$$

3. The filters required should cover a total bandwidth which is a significant part of the maximum unambiguous bandwidth (the sampling frequency).

Restriction (1) means that the amplitude part of the weighting sequence is the same for each filter and allows the weights to be the roots of a unit vector with a zero phase angle. Then the amplitude sequence need be applied only once to the incoming data sequence for the entire filter bank. The weights that distinguish different filters in the bank all have the same magnitude.

If we assume that a table is available to supply the real and imaginary parts of $e^{-i2\pi n/N}$, the major arithmetic (to evaluate Equation (A-2) in order to synthesize a filter bank) is a series of complex multiplications and additions. Using the base-two (or $B = 2$) version of the fast Fourier method, one of each two input measurements (vector voltages) is rotated by an appropriate amount and added to and also subtracted from the other of a pair of vector measurements.

TABLE A–3—Second Step of Fast Fourier or Cooley-Tukey Process

(16 samples per frame: B = Base = 2, T = Result of first step, U = Result of second step)

$T_1 + T_5 = U_1$	
$T_2 + T_6 = U_2$	$e^{i0} = 1 + i0 = 1$
$T_3 + T_7 = U_3$	$e^{i\pi/2} = 0 + i1 = i$
$T_4 + T_8 = U_4$	$e^{i2\pi/2} = -1 + i0 = -1$
$T_1 - T_5 = U_5$	$e^{i3\pi/2} = 0 - i1 = -i$
$T_2 - T_6 = U_6$	
$T_3 - T_7 = U_7$	
$T_4 - T_8 = U_8$	
$T_9 + T_{13}(i) = U_9$	
$T_{10} + T_{14}(i) = U_{10}$	
$T_{11} + T_{15}(i) = U_{11}$	
$T_{12} + T_{16}(i) = U_{12}$	
$T_9 - T_{13}(i) = U_{13}$	
$T_{10} - T_{14}(i) = U_{14}$	
$T_{11} - T_{15}(i) = U_{15}$	
$T_{12} - T_{16}(i) = U_{16}$	

Thus, for 16 input samples, there would be 8 "pairs." These are the first and the ninth, the second and tenth, the third and eleventh, and so on. Rotations at this step are by multiples of $2\pi/2$ radians.

The procedures for the first step are described by Table A-2 in which S's are samples and T's are the result of the processing.

During the second step, the T's are operated upon in a similar manner except that this time the T's taken together are four subscripts apart instead of eight as were the S's. (The general rule is that the S's are one-half the sample set apart, the T's one-quarter the sample set apart and subsequently the U's which come from the T's will be one-eighth set apart and so on.) Also, during the second step, the vector rotations are by multiples of $2\pi/4$ radians and accordingly the vector multiplications are by:

$$e^{n\pi/2} = \cos\frac{n\pi}{2} + i\sin\frac{n\pi}{2} = \begin{cases} (-1)^{n/2} & n = \text{even} \\ i(-1)^{n-1/2} & n = \text{odd} \end{cases}$$

In each subsequent step, the vector rotations are by multiples of angles one-half as great as in the preceding step. Thus to form V's from U's, rotations are by multiples of $\pi/4$ and the sines and cosines are 0, ± 1, ± 0.707; to form X's which are from the V's, the rotations are by multiples of $\pi/8$ radians. The X's differ from the filter outputs, or F's only in the permutation of subscripts (see Table A–5).

The procedures for the second, third, and fourth (last) steps are shown in Tables A–3, A–4, and A–5. It should be noted that for the case shown, there are 4 steps. This is true because using the base, B of 2, there are $\log_B 16 = 4$ steps as was previously described. And each step required $N/B = N/2$ complex multiplications and $N/B = N/2$ complex additions as was described.

The example given here is perhaps trivial because only 16 samples were used for each data frame and because the base used was only 2. However, if the same 16 samples were used but the base were raised to 4, the number of steps would reduce from 4 to 2 with an overall reduction ratio of $16\log_2 16 : 16\log_4 16$ or 2:1. In the case of a base-4 approach rotations in the first step would be by multiples of $2\pi/4$ and the first, fifth, ninth, and thirteenth sample would be summed after being rotated, etc. In the second and last step, rotations would be by multiples of $2\pi/16$ and the first, second, third and fourth result of the first step would be summed after being rotated, etc. Thus, the extension of the simple-16 sample base-2 example to the equally simple 16-sample base-4 example shows the general method of the fast Fourier method as well as the great reduction in processing compared to the conventional method.

TABLE A–4—Third Step of Fast Fourier or Cooley-Tukey Process

(16 samples per frame: B = Base = 2, U = Result of second step, V = Result of third step)

$U_1 + U_3 = V_1$	
$U_2 + U_4 = V_2$	$e^{i0} = 1 + i0 = 1$
$U_1 - U_3 = V_3$	$e^{i\pi/4} = 0.707 + i0.707$
$U_2 - U_4 = V_4$	$e^{i2\pi/4} = 0 + i1 = i$
$U_5 + U_7(i) = V_5$	$e^{i3\pi/4} = -0.707 + i0.707$
$U_6 + U_8(i) = V_6$	$e^{i4\pi/4} = -1 + i0 = -1$
$U_5 - U_7(i) = V_7$	$e^{i5\pi/4} = -0.707 - i0.707$
$U_6 - U_8(i) = V_8$	$e^{i6\pi/4} = 0 - i1 = -i$
$U_9 + U_{11}(0.707 + i0.707) = V_9$	$e^{i7\pi/4} = 0.707 - i0.707$
$U_{10} + U_{12}(0.707 + i0.707) = V_{10}$	
$U_9 - U_{11}(0.707 + i0.707) = V_{11}$	
$U_{10} - U_{12}(0.707 + i0.707) = V_{12}$	
$U_{13} + U_{15}(-0.707 + i0.707) = V_{13}$	
$U_{14} + U_{16}(-0.707 + i0.707) = V_{14}$	
$U_{13} - U_{15}(-0.707 + i0.707) = V_{15}$	
$U_{14} - U_{16}(-0.707 + i0.707) = V_{16}$	

TABLE A-5—Last (Fourth) Step of Fast Fourier or Cooley-Tukey Process

(16 samples per frame B = Base = 2, U = Result of third step, F = Filtered outputs = Result of fourth step)

$V_1 + V_2 = X_1 = F_0$	$e^{i0} = 1 + i0 = 1$
$V_1 - V_2 = X_2 = F_8$	$e^{i\pi/8} = 0.924 + i0.383$
$V_3 + V_4(i) = X_3 = F_4$	$e^{i2\pi/8} = 0.707 + i0.707$
$V_3 - V_4(i) = X_4 = F^{12}$	$e^{i3\pi/8} = 0.383 + i0.924$
$V_5 + V_6(0.707 + i0.707) = X_5 = F_2$	$e^{i4\pi/8} = 0 + i = i$
$V_5 - V_6(0.707 + i0.707) = X_6 = F_{10}$	$e^{i5\pi/8} = -0.383 + i0.984$
$V_7 + V_8(-0.707 + i0.707) = X_7 = F_6$	$e^{i6\pi/8} = -0.707 + i0.707$
$V_7 - V_8(-0.707 + i0.707) = X_8 = F_{14}$	$e^{i7\pi/8} = -0.924 + i0.383$
$V_9 + V_{10}(0.924 + i0.383) = X_9 = F_1$	$e^{i8\pi/8} = -1 + i0 = -1$
$V_9 - V_{10}(0.924 + i0.383) = X_{10} = F_5$	$e^{i9\pi/8} = -0.924 - i0.383$
$V_{11} + V_{12}(-0.383 + i0.924) = X_{11} = F_9$	$e^{i10\pi/8} = -0.707 - i0.707$
$V_{11} - V_{12}(-0.383 + i0.924) = X_{12} = F_{13}$	$e^{i11\pi/8} = -0.383 - i0.924$
$V_{13} + V_{14}(0.383 + i0.924) = X_{13} = F_3$	$e^{i12\pi/8} = 0 - i = -i$
$V_{13} - V_{14}(0.383 + i0.924) = X_{14} = F_{11}$	$e^{i13\pi/8} = 0.383 - i0.984$
$V_{15} + V_{16}(-0.924 + i0.383) = X_{15} = F_7$	$e^{i14\pi/8} = 0.707 - i0.707$
$V_{15} - V_{16}(-0.924 + i0.383) = X_{16} = F_{15}$	$e^{i15\pi/8} = 0.924 - i0.383$

An improved generalized inverse algorithm for linear inequalities and its applications

by L. C. GEARY* and C. C. LI

University of Pittsburgh
Pittsburgh, Pennsylvania

INTRODUCTION

A great amount of research for the solution of linear inequalities has been undertaken in the past ten years. One of the reasons for this research is the development of linear separation approaches to pattern recognition[1-5,8-16] and threshold logic problems.[6,7,9] Both of these problems require the determination of a decision function or decision functions which, in the case of linear separation, involve a system of linear inequalities.

In this paper, an improved iterative algorithm will be developed for the solution of the set of linear inequalities which is written in the following equation:

$$Aw > 0. \tag{1}$$

This algorithm is an improvement of the Ho-Kashyap algorithm by choosing a criterion function

$$J(y) = 4 \sum_{i=1}^{N} (\cosh \tfrac{1}{2} y_i)^2 \tag{2}$$

to be minimized where y_i is the ith component of the N by 1 vector y defined below

$$y = Aw - b, b > 0. \tag{3}$$

The improvement lies in an acceleration of the Ho-Kashyap algorithm caused by a steeper gradient of $J(y)$ as can be seen when a comparison is made between the two criterion functions. Let $J_{hk}(y)$ designate the criterion function used in the Ho-Kashyap algorithm,

$$J_{hk}(y) = \| y \|^2 = \sum_{i=1}^{N} y_i^2. \tag{4}$$

Since $J(y)$ and $J_{hk}(y)$ reach their respective minimum when each $(\cosh \frac{1}{2} y_i)^2$ and each y_i^2 are respectively minimized, one can simply compare $J(y_i)$ and $J_{hk}(y_i)$, the convex functions of one variable only. Taking the gradients of $J(y_i)$ and $J_{hk}(y_i)$ with respect to y_i, one obtains

$$\frac{\partial J(y_i)}{\partial y_i} = 2y_i + \frac{2}{3!} y_i^3 + \frac{2}{5!} y_i^5 + \cdots \tag{5}$$

and

$$\frac{\partial J_{hk}(y_i)}{\partial y_i} = 2y_i. \tag{6}$$

It is clear that the absolute value of $\partial J(y_i)/\partial y_i$ is greater than the absolute value of $\partial J_{hk}(y_i)/\partial y_i$ everywhere except at $y_i = 0$ where they are equal. In general, the gradient $\partial J(y)/\partial y$ is greater than the gradient $\partial J_{hk}(y)/\partial y$ everywhere except at the origin $y = 0$. Since the gradient descent procedure is used in both algorithms, and since y and b, or y and w, are linearly related, it is conceivable that the proposed algorithm may have a higher convergence rate for a solution w.

As mentioned previously, $J(y)$ reaches a minimum when each term $(\cosh \frac{1}{2} y_i)^2$, $(i = 1, \ldots, N)$, is minimized. For each $(\cosh \frac{1}{2} y_i)^2$ to be a minimum, each y_i, $(i = 1, \ldots, N)$, must equal zero and $y = 0$ gives a desired solution. Since the b_i's are only constrained to be positive, $J(y)$ can be minimized with respect to both w and b subject to the condition that $b > 0$. Note that it is not necessary to attain the minimum value of $J(y)$; in fact, a solution w^* is obtained whenever $y \geq 0$ with $b > 0$ from which follows $Aw^* \geq b > 0$.

DEVELOPMENT OF THE TWO-CLASS ALGORITHM

Let the matrix A, whose transpose is

$$A^t = [{}_1x_1, \ldots, {}_{n_1}x_1, {}_1x_2, \ldots, {}_{n_2}x_2]$$

*Presently with Gulf Research & Development Company, Pittsburgh, Pa.

be represented as

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdot & \cdot & \cdots & \cdot \\ a_{N1} & a_{N2} & \cdots & a_{Nn} \end{bmatrix}, \tag{7}$$

where ${}_ix_i$ is an n by 1 augmented pattern vector, $n = r + 1$, and $N = n_1 + n_2$. The gradient of $J(y)$ with respect to w is given by

$$\frac{\partial J(y)}{\partial w} = 2A^t s(y) \tag{8}$$

where

$$s^t(y) = [\sinh y_1, \cdots, \sinh y_N],$$

and the gradient of $J(y)$ with respect to b is given by

$$\frac{\partial J(y)}{\partial b} = -2s(y), \tag{9}$$

where the derivative of a scalar with respect to a column vector is a column vector. Since w is not constrained in any way $\partial J(y)/\partial w = 0$ implies $s(y) = 0$ which, in turn, implies $y_i = 0$ for all $i = 1, 2, \cdots, N$. Therefore, for a fixed $b > 0$, minimizing $J(y)$ with respect to w gives

$$y = Aw - b = 0.$$

Solving the above equation for w, one obtains

$$w = A^{\#}b \tag{10}$$

where $A^{\#}$ is the generalized inverse of A.

On the other hand, for a fixed w, $\partial J(y)/\partial b = 0$ with $b > 0$ dictates a descent procedure of the following form, with k denoting the iteration number:

$$b(k+1) = b(k) + \Delta b(k) \tag{11}$$

where the components of $\Delta b_i(k)$, $i = 1, 2, \cdots, N$, of $\Delta b(k)$ are governed by

$$\Delta b_i(k) \propto \begin{cases} -\left(\dfrac{\partial J(y(k))}{\partial b}\right)_i = 2 \sinh y_i & \text{if } y_i > 0, \\ 0 & \text{if } y_i \leq 0. \end{cases} \tag{12}$$

Introduce a positive scalar $p(k)$ as the proportionality constant and rewrite equation (12) in the vector form,

$$\Delta b(k) = p(k)h(k), \tag{13}$$

where

$$h(k) = [h_i(k)] = [\sinh y_i(k) + |\sinh y_i(k)|] \quad (i = 1, 2, \cdots, N). \tag{14}$$

As can be shown later, $p(k)$ may be chosen to equal

$$p(k) = \frac{1}{\cosh y_{\max}(k)} \tag{15}$$

where

$$y_{\max}(k) = \max_i |y_i(k)|. \tag{16}$$

Substituting (13) into (11) and, from (10), writing

$$w(k+1) = w(k) + p(k)A^{\#}h(k), \tag{17}$$

one obtains the following algorithm:

$$\begin{cases} w(0) = A^{\#}b(0), b(0) > 0 \text{ but otherwise arbitrary} \\ y(k) = Aw(k) - b(k) \\ b(k+1) = b(k) + p(k)h(k) \\ w(k+1) = w(k) + p(k)A^{\#}h(k) \end{cases} \tag{18}$$

where $h(k)$ and $p(k)$ are given by equations (14) and (15) respectively. Note that in this algorithm $p(k)$ varies at each step and is a nonlinear function of $y(k)$. A recursive relation in $y(k)$ can also be obtained from (18),

$$y(k+1) = y(k) + p(k)(AA^{\#} - I)h(k). \tag{19}$$

Just like the Ho-Kashyap algorithm, it can be shown that the above algorithm (18) converges to a solution w^* of the system of linear inequalities in a finite number of steps provided that a solution exists, and simultaneously acts as a test for the inconsistency of the linear inequalities. These properties are formally stated in Theorem I given in the next section.

THEOREM I

Before discussing the main theorem, a lemma to be used in the proof of the theorem will be given first.

Lemma 1: Let one consider the set of linear inequalities (1) and the algorithm (18) to solve this set. Then

1) $y(k) \not> 0$ for any k;

and

2) if the set of linear inequalities is consistent, then

$$y(k) \not< 0 \quad \text{for any } k.$$

This lemma is the same as the one given by Ho and Kashyap[8] except that the iterative algorithm is different. The proof of the lemma is not given here since it is similar to the proof of Ho-Kashyap lemma. Recall again the notation used in the lemma: $y(k) \leq 0$ means that $y_i(k) \leq 0$ for all i but y possesses at least one negative component. This lemma is a rigorous state-

ment that with a consistent set of linear inequalities $Aw > 0$, the elements of the vector $y(k)$ cannot be all non-positive.

Theorem I: Consider the set of linear inequalities (1) and the algorithm (18) to solve these inequalities, and let $V[y(k)] = \| y(k) \|^2$.

1) If the set of linear inequalities is consistent then
   a) $\Delta V[y(k)] \triangleq V[y(k+1)] - V[y(k)] < 0$ and $\lim_{k\to\infty} V[y(k)] = 0$ implying convergence to a solution in an infinite number of steps; and
   b) actually, a solution is obtained in a finite number of steps.
2) If the set of linear inequalities is inconsistent, then there exists a positive integer k^* such that

$$\Delta V[y(k)] < 0 \quad \text{for} \quad k < k^*$$
$$\Delta V[y(k)] = 0 \quad \text{for} \quad k \geq k^*,$$

and

$$y(k) \not< 0 \quad \text{for} \quad k < k^*$$
$$y(k) = y(k^*) \leq 0 \quad \text{for} \quad k \geq k^*$$

and

$$w(k) = w(k^*) \quad \text{for} \quad k \geq k^*$$
$$b(k) = b(k^*) \quad \text{for} \quad k \geq k^*.$$

In other words, the occurrence of a nonpositive vector $y(k)$ at any step terminates the algorithm and indicates the inconsistency of the given set of linear inequalities.

Proof:

Part 1: Since the algorithm (18) can be rewritten as a recursive relation in $y(k)$ given by (19), and[8]

$$V[y(k)] = \| y(k) \|^2 > 0 \quad \text{for all} \quad y(k) \neq 0 \tag{20}$$

$V[y(k)]$ can be considered as a Liapunov function for the nonlinear difference equation (19). Thus

$$\begin{aligned}\Delta V[y(k)] &\triangleq V[y(k+1)] - V[y(k)] \\ &= \| y(k+1) \|^2 - \| y(k) \|^2 \\ &= y^t(k+1)y(k+1) - y^t(k)y(k) \\ &= p(k)h^t(k)(AA^\# - I)^t y(k) \\ &\quad + p(k)y^t(k)(AA^\# - I)h(k) \\ &\quad + p^2(k)h^t(k)(AA^\# - I)^t(AA^\# - I)h(k).\end{aligned}$$

Since $(AA^\# - I)$ is hermitian idempotent, and $AA^\# y(k) = 0$, $\Delta V[y(k)]$ reduces to

$$\begin{aligned}\Delta V[y(k)] &= -2p(k)h^t(k)y(k) \\ &\quad + p^2(k)h^t(k)(I - AA^\#)h(k). \end{aligned}\tag{21}$$

Further simplification leads to

$$\begin{aligned}\Delta V[y(k)] &= -[y(k) + |\, y(k)\,|\,]^t[p(k)R(k) \\ &\quad + p^2(k)R(k)(AA^\# - I)R(k)][y(k) + |\, y(k)\,|\,] \\ &= -\| y(k) + |\, y(k)\,|\,\|^2_{p^2(k)R(k)AA^\#R(k)} \\ &\quad + p(k)R(k) - p^2(k)R^2(k). \end{aligned}\tag{22}$$

where $R = \operatorname{diag}\left[\dfrac{\sinh y_1}{y_1}, \cdots, \dfrac{\sinh y_N}{y_N}\right]$.

For $\Delta V[y(k)]$ to be negative semidefinite, $\Delta V[y(k)] = 0$ only if $y(k) = 0$ or $y(k) \leq 0$, the matrix

$$[p^2(k)R(k)AA^\#R(k) + p(k)R(k) - p^2(k)R^2(k)]$$

must be positive definite. $AA^\#$ is positive semidefinite because $AA^\#$ is hermitian idempotent, $x^tAA^\#x \geq 0$ for any x; it follows that $z^tRAA^\#Rz \geq 0$ for any z; hence $RAA^\#R$ is also positive semidefinite. Now one can choose a $p(k)$ such that $[p(k)R(k) - p^2(k)R^2(k)]$ is positive definite. $[p(k)R(k) - p^2(k)R^2(k)]$ is positive definite if

$$[p(k)r_{ii}(k) - p^2(k)r_{ii}^2(k)] > 0$$
$$\text{for all} \quad i = 1, 2, \cdots, N. \tag{23}$$

Since $r_{ii}(k) = \sinh y_i / y_i > 0$ for all i and $p(k)$ is restricted to be positive, the above condition reduces to the condition,

$$1 - p(k)r_{ii}(k) > 0 \quad \text{for all} \quad i = 1, 2, \cdots, N. \tag{24}$$

For $p(k)$ chosen in equation (15),

$$p(k) = \frac{1}{\cosh y_{\max}(k)}$$

$$p(k)r_{ii}(k) = \frac{1}{\cosh y_{\max}(k)} \frac{\sinh y_i(k)}{y_i(k)}$$

$$= \frac{\sinh y_i(k)}{y_i(k) \cosh y_{\max}(k)}$$

$$= \frac{\displaystyle\sum_{n=0}^{\infty} \frac{y_i^{2n}(k)}{(2n+1)!}}{\displaystyle\sum_{n=0}^{\infty} \frac{y_{\max}^{2n}(k)}{(2n)!}} < 1.$$

Thus the condition (24) is satisfied and $[p(k)R(k) -$

$p^2(k)R^2(k)]$ is positive definite for

$$p(k) = \frac{1}{\cosh y_{\max}(k)}.$$

Then $\Delta V[y(k)]$ has the desired property of negative semidefinite for $p(k) = 1/\cosh y_{\max}(k)$ and for any finite $y(k)$.

From equation (22) one notes that $\Delta V[y(k)]$ equals zero if and only if $y(k) = 0$ or $y(k) \leq 0$. Since it is assumed that the set of linear inequalities (1) is consistent, and from the lemma $y(k) \not< 0$, therefore

$$\begin{aligned} \Delta V[y(k)] &< 0 \quad \text{for all} \quad y(k) \neq 0 \\ &= 0 \quad \text{if} \quad y(k) = 0. \end{aligned} \tag{25}$$

By Liapunov's stability criterion, the equilibrium state $y = 0$ of the discrete system (19) can be reached asymptotically, i.e., $\lim_{k\to\infty} \| y(k) \|^2 = 0$, which corresponds to a solution w^{**} with $Aw^{**} = b > 0$. This completes the proof of Part 1(a).

To prove the convergence of the algorithm (18) in a finite number of steps, one notes that $b(k)$ is a nondecreasing vector. Let $b^t(0) = [1, 1, \cdots, 1]$, then

$$b^t(k) \geq b^t(0) \geq [1, 1, \cdots, 1] \quad \text{for any } k > 0.$$

Since $Aw(k) = b(k) + y(k)$, $| y^t(k) | < [1, 1, \cdots, 1]$ implies $Aw^*(k) > 0$ when a solution w^* is reached. But $V[y(k)] \leq 1$ implies $| y^t(k) | < [1, 1, \cdots, 1]$. Since $V[y(k)]$ converges to zero in infinite time, it must converge to the region $V[y(k)] = 1$ in finite time, hence $| y^t(k) | < [1, 1, \cdots, 1]$, $Aw(k) > 0$, and a solution $w^* = w(k)$ is obtained in a finite number of steps. This completes the proof of Part 1(b).

Part 2: It has been proved in Part 1 that $V[y(k)]$ is negative semidefinite independent of the consistency of the linear inequalities. Now, if the set of linear inequalities (1) is inconsistent, one notes that $y(k)$ cannot be 0 and hence $V[y(k)]$ cannot become zero for any $k > 0$. There must exist a value of k, called k^*, such that

$$\begin{aligned} \Delta V[y(k)] &< 0 \quad \text{for} \quad 0 \leq k < k^* \\ &= 0 \quad \text{for} \quad k = k^*, \\ y(k) &\not< 0 \quad \text{for} \quad 0 \leq k < k^*. \end{aligned}$$

But $V[y(k^*)] = 0$ if either $y(k^*) = 0$ or $y(k^*) \leq 0$. Since $y(k^*) \neq 0$, this implies $y(k^*) \leq 0$ and hence, from (14), $h(k^*) = 0$. Equation (19) indicates that

$$y(k) = y(k^*) \leq 0 \quad \text{for all} \quad k \geq k^*$$

As a consequence, one obtains

$$\begin{aligned} \Delta V[y(k)] &= 0 \quad \text{for all} \quad k \geq k^* \\ h(k) &= 0 \quad \text{for all} \quad k \geq k^* \\ w(k) &= w(k^*) \quad \text{for all} \quad k \geq k^* \\ b(k) &= b(k^*) \quad \text{for all} \quad k \geq k^* \end{aligned}$$

This completes the proof of the theorem.

An Optimum Choice of the Scalor $p(k)$

The choice of $p(k) = 1/\cosh y_{\max}(k)$ in the previous section is only one of many possible choices of $p(k)$ for the convergence of the algorithm (18). The convergence rate may be further improved by choosing a $p(k)$ such that the decrease in the Lyapunov function $V[y(k)]$ is maximized at every step, that is, $-\Delta V[y(k)]$ is maximized with respect to $p(k)$. Taking the partial derivative of $\Delta V[y(k)]$ in equation (22) with respect to $p(k)$ leads to an optimum value of $p(k)$ given by

$$p(k) = \frac{[y(k) + | y(k) |]^t R(k)[y(k) + | y(k) |]}{2[y(k) + | y(k) |]^t R(k) \cdot [I - AA^{\#}]R(k)[y(k) + | y(k) |]} \tag{26}$$

provided that $I - AA^{\#} > 0$. For this value of $p(k)$, $\Delta V[y(k)]$ is negative definite in $[y(k) + | y(k) |]$ which is required in the convergence proof of the algorithm (18). A flow chart summarizing the above procedure is shown in Figure 1.

EXAMPLES

The algorithm (18) has been applied to pattern recognition and switching theory problems. For switching theory problems the generalized inverse of the N by n pattern matrix A is simplified to

$$A^{\#} = 2^{-(n-1)}A^t.$$

Two example problems will be presented, one in switching theory and the other in pattern recognition.

Example 1: Consider a Boolean function of eight binary variables which corresponds to the separation of the two classes:

Class C_1 = (127, 191, 215, 217 to 255)

Class C_2 = (0 to 126, 128 to 190, 192 to 214, 216).

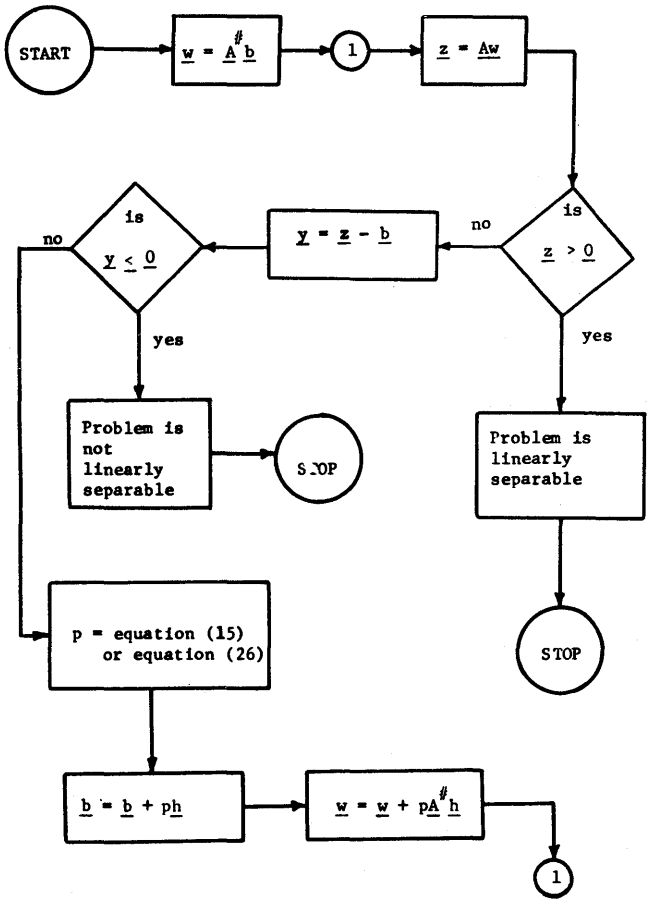


Figure 1—Flow chart of the proposed 2-class algorithm

Here $m = 2^r = 256$ and $n = r + 1 = 9$, where r is the number of binary variables. For

$$b^t(0) = [.1, .1, .1, \cdots, .1, .1, .1]$$

and $p(k)$ given in equation (26), the algorithm terminates after the tenth iteration and gives a solution weight vector w for the switching function,

$$w^t = [0.3732, 0.2278, 0.2278, 0.1654, 0.0769, 0.0569, 0.0247, 0.0247, 0.0247],$$

The same example was solved using the Ho-Kashyap algorithm.[8] It required 229 iterations with the same initial $b(0)$. The solution weight vector w for the Ho-Kashyap algorithm is

$$w^t = [0.5741, 0.3447, 0.3447, 0.2425, 0.1155, 0.1080, 0.0436, 0.0436, 0.0436].$$

The computing time for the proposed algorithm was 50 seconds on IBM 7090 with a cost of $1.50, while the Ho-Kashyap algorithm required 80 minutes with a cost of $23.50. Thus the proposed algorithm not only reduced the number of required iterations but also the computing time and cost to solve the problem. It was observed, that for $0.5 \geq b_i(0) \geq 0.001$ and $p(k)$ given by equation (26), for all examples tried by the authors that the number of iterations was less than or equal to the number of iterations required by the Ho-Kashyap algorithm. In some cases the number of iterations was reduced by a factor of 25.[17]

Example 2: The proposed algorithm was also applied to a preliminary study of a biomedical pattern recognition problem. The problem is to investigate whether or not a change exists in the diurnal cycle of an individual person upon a change in his environmental condition or physiological state and if such a change may be used to diagnose physical ailments under strictly controlled conditions by measuring the amounts of electrolytes present in urine samples every three hours.[18] The data used in this example consisted of thirteen sample patterns under two different conditions. Each pattern has eight components which represent the mean excretion rates of an electrolyte for each three-hour period of the twenty-four hour cycle. Thus $N = 13$ and $n = r + 1 = 8 + 1 = 9$; the size of the pattern matrix A is 13 by 9. The pattern matrix A is shown in Table 1. Let $b^t(0) = [0.1, 0.1, \cdots, 0.1]$. For this problem the Ho-Kashyap algorithm with $p = 1$ required 7 iterations to determine the separability. However, the proposed algorithm with $p(k)$ given by equation (26) required only two iterations, where $p(1) = 5.270684$ and $p(2) = 3.197152$. The problem is linearly separable and a solution weight vector w obtained by the proposed algorithm is

$$w^t(2) = [-13.6089, 2.5915, 1.6847, 2.2314, 0.3414, 3.0077, 1.8428, 1.6559, 0.0096]$$

EXTENSION TO THE MULTICLASS ALGORITHM

The problem of multiclass patterns classification is that it must be determined to which of the R different classes, $C_1, C_2, \cdots, C_R$, a given pattern vector, x, belongs. If the R-class patterns are linearly separable, there exist R weight vectors w_j to construct R discriminant functions $g_j(x)$, $(j = 1, 2, \cdots, R)$, such that

$$g_j(x) = x^t w_j > x^t w_i = g_i(x) \text{ for all } i \neq j, \; x \in C_j. \quad (27)$$

Chaplin and Levadi[10] have formulated another set of inequalities which can be considered as a representation of linear separation of R-class patterns. This set of inequalities is

$$\| x^t U - e_j^t \| < \| x^t U - e_i^t \| \text{ for all } i \neq j, \; x \in C_j \quad (28)$$
$$\text{for all } j = 1, 2 \cdots, R$$

TABLE 1—The Pattern Matrix A for Example 2

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
1.00	.96	1.19	1.35	.75	1.12	.94	.73	.97
1.00	.75	1.19	1.35	1.06	1.07	.97	.81	.81
1.00	.80	1.13	.85	.90	1.14	1.27	1.01	.88
1.00	.66	1.40	1.25	1.09	1.54	.79	.27	.00
1.00	2.04	1.14	1.10	.57	.62	.66	.47	1.39
1.00	1.02	1.32	1.06	1.03	1.07	1.16	.77	.57
−1.00	−.48	−1.01	−.68	−.72	−1.76	−1.25	−.62	−1.47
−1.00	−.55	−.55	−1.04	−.91	−1.40	−1.17	−1.28	−1.09
−1.00	−.87	−.79	−1.34	−.86	−.44	−2.15	−.82	−.74
−1.00	−.09	−.70	−.67	−.80	−1.93	−1.29	−1.14	−1.39
−1.00	−1.12	−1.75	−.51	−.72	−1.25	−.46	−.89	−1.29
−1.00	−1.20	−1.47	−.60	−.96	−1.13	−.89	−.74	−1.00
−1.00	−1.43	−1.79	−.68	−.75	−.82	−.56	−.94	−1.04

where U is an $n \times (R-1)$ weight matrix and the vectors e_j's are the vertex vectors of a $R-1$ dimensional equilateral simplex with its centroid at the origin. If each e_j is associated with one class, x is classified according to the nearest neighborhood of the mapping x^tU to the vertices. Inequalities (28) are, in fact, equivalent to inequalities (27) with

$$w_j = Ue_j, \qquad (j = 1, 2, \cdots, R) \tag{29}$$

Let the $N \times n$ pattern matrix A be defined in the following manner,

$$A = \begin{bmatrix} A_1 \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ A_j \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ A_R \end{bmatrix} \triangleq \begin{bmatrix} {}_1x^t{}_1 \\ \cdot \\ \cdot \\ \cdot \\ n_1x^t{}_1 \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1x^t{}_j \\ \cdot \\ \cdot \\ \cdot \\ n_jx^t{}_i \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1x^t{}_R \\ \cdot \\ \cdot \\ \cdot \\ n_Rx^t{}_R \end{bmatrix} \tag{30}$$

where A_j is an $n_j \times n$ submatrix having as its rows n_j transposed pattern vectors of class C_j,

$${}_lx^t{}_j, \qquad (l = 1, 2, \cdots, n_j),$$

where the right subscript denotes the pattern class and the left subscript denotes the lth pattern in that class, and $N = n_1 + n_2 + \cdots + n_R$. Designate the $n \times (R-1)$ weight matrix U as composed of $(R-1)$ column vectors u_q, $(q = 1, 2, \cdots, R-1)$,

$$U = [u_1 \cdots u_q \cdots u_{R-1}]. \tag{31}$$

Also define an $N \times (R-1)$ matrix B as

$$B = \begin{bmatrix} B_1 \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ B_j \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ B_R \end{bmatrix} \triangleq \begin{bmatrix} {}_1b^t{}_1 \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_1}b^t{}_1 \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1b^t{}_j \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_j}b^t{}_j \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1b^t \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_R}b^t{}_R \end{bmatrix} \tag{32}$$

whose row vectors ${}_lb^t_j$, ($j = 1, 2, \cdots, R; l = 1, 2, \cdots, n_j$), correspond to the class groupings in the A matrix and satisfy the following inequalities

$$ {}_lb^t_j(e_j - e_i) > 0 \qquad \text{for all } i \neq j \tag{33} $$
$$ \text{for all } j = 1, 2, \cdots, R. $$

B_j is an $n_j \times (R-1)$ submatrix of B, $j = 1, 2, \cdots, R$. Let an $N \times (R-1)$ matrix Y be defined as

$$ Y \triangleq AU - B. \tag{34} $$

The representation of Y may be in the form of either an array of $(R-1)$ column vectors, y_q, ($q = 1, 2, \cdots, R-1$),

$$ Y = [y_1 \cdots y_q \cdots y_{R-1}] \tag{35} $$

or an array of N row vectors ${}_lY_j$, ($j = 1, 2, \cdots, R$; $l = 1, 2, \cdots, n_j$), corresponding to the class groupings in the A matrix,

$$ Y = \begin{bmatrix} Y_1 \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ Y_j \\ -- \\ \cdot \\ \cdot \\ \cdot \\ -- \\ Y_R \end{bmatrix} \triangleq \begin{bmatrix} {}_1Y_1 \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_1}Y_1 \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1Y_j \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_j}Y_j \\ --- \\ \cdot \\ \cdot \\ \cdot \\ --- \\ {}_1Y_R \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_R}Y_R \end{bmatrix} \tag{36} $$

where Y_j is an $n_j \times (R-1)$ submatrix of Y,

$$ Y_j = A_jU - B_j \tag{37} $$

or

$$ {}_lY_j = {}_lx^t_jU - {}_lb^t_j \qquad j = 1, 2, \cdots, R $$
$$ l = 1, 2, \cdots, n_j. $$

The set of linear inequalities which will be discussed in this paper is

$$ A_jU(e_j - e_i) > 0 \qquad \text{for all } i \neq j \tag{38} $$
$$ \text{for all } j = 1, 2, \cdots, R $$

Associated with it is another set of linear inequalities

$$ Y_j(e_j - e_i) = (A_jU - B_j)(e_j - e_i) > 0 \tag{39} $$
$$ \text{for all } i \neq j $$
$$ \text{for all } j = 1, 2, \cdots, R $$

or

$$ {}_lY_j(e_j - e_i) = ({}_lx^t_jU - {}_lb^t_j)(e_j - e_i) > 0 $$
$$ \text{for all } i \neq j $$
$$ \text{for all } j = 1, 2, \cdots R $$
$$ \text{for all } l = 1, 2, \cdots n_j. $$

Since, by (33), $B_j(e_j - e_i)$ is constrained to have positive components for all $i \neq j$, inequalities (39) implies the inequalities (38) and hence (27) or (28). When inequalities (38) are satisfied for all $i \neq j$ and for all $j = 1, 2, \cdots, R$, a solution weight matrix U is reached which will give linear classification of R-class patterns; that is, if

$$ x^tU(e_j - e_i) > 0 \qquad \text{for all } i \neq j $$

then x is classified as of class C_j.

DEVELOPMENT OF THE MULTI-CLASS ALGORITHM

For the notational simplicity in the derivation of the gradient function to be developed below, let the matrices A, U, B, and Y in equations (30), (31), (32), and (35) be represented respectively as

$$ A = \begin{bmatrix} a_{11} & a_{12} & a_{1n} \\ \cdots & \cdots & \cdots \\ a_{N1} & a_{N2} & a_{Nn} \end{bmatrix} \tag{40} $$

$$ U = \begin{bmatrix} u_{11} & u_{12} & u_{1,R-1} \\ \cdots & \cdots & \cdots \\ u_{n1} & u_{n2} & u_{n,R-1} \end{bmatrix} \tag{41} $$

$$ B = \begin{bmatrix} b_{11} & b_{12} & b_{1,R-1} \\ \cdots & \cdots & \cdots \\ b_{N1} & b_{N2} & b_{N,R-1} \end{bmatrix} \tag{42} $$

and

$$ Y = \begin{bmatrix} y_{11} & y_{12} & y_{1,R-1} \\ \cdots & \cdots & \cdots \\ y_{N1} & y_{N2} & y_{N,R-1} \end{bmatrix} \tag{43} $$

Substituting these into equation (34), one obtains

$$y_{ij} = \sum_{k=1}^{n} a_{ik}u_{kj} - b_{ij}. \tag{44}$$

Let $C(Y)$ be an $N \times (R-1)$ matrix defined by

$$C(Y) = [c_{ij}] \triangleq [\cosh \tfrac{1}{2}y_{ij}] \tag{45}$$
$$(i = 1, \cdots, N; j = 1, \cdots, R-1).$$

The criterion function $J(Y)$ to be minimized is chosen as the trace of $4C^t(Y)C(Y)$,

$$J(Y) \triangleq \text{Tr}\,(4C^tC) = \sum_{i=1}^{N}\sum_{j=1}^{R-1} J_{ij}(Y) \tag{46}$$

where

$$J_{ij}(Y) = 4\,(\cosh \tfrac{1}{2}y_{ij})^2.$$

Determine the gradients of $J(Y)$ with respect to both U and B,

$$\frac{\partial J(Y)}{\partial U} = 2A^tS(Y) \tag{47}$$

$$\frac{\partial J(Y)}{\partial B} = -2S(Y) \tag{48}$$

where $S(Y)$ is an $N \times (R-1)$ matrix with the following representation

$$S(Y) \triangleq [\sinh y_{ij}]$$
$$(i = 1, 2, \cdots, N; j = 1, \cdots, R-1)$$

$$\triangleq \begin{bmatrix} S_1(Y) \\ \text{-----} \\ S_j(Y) \\ \text{-----} \\ \cdot \\ \cdot \\ \cdot \\ \text{-----} \\ S_R(Y) \end{bmatrix} \triangleq \begin{bmatrix} {}_1S_1(Y) \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_1}S_1(Y) \\ \text{------} \\ {}_1S_j(Y) \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_j}S_j(Y) \\ \text{------} \\ \cdot \\ \cdot \\ \cdot \\ \text{------} \\ {}_1S_R(Y) \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_R}S_R(Y) \end{bmatrix} \tag{49}$$

and ${}_lS_j(Y)$ is a row vector of the following form

$${}_lS_j(Y) = [{}_lS_{j1}(Y),\ {}_lS_{j2}(Y), \cdots,\ {}_lS_{j(R-1)}(Y)]$$
$$= [\sinh y_{(n_{j-1}+l),1}, \cdots, \sinh y_{(n_{j-1}+l),R-1}]. \tag{50}$$

Since U is not constrained in any manner, $\partial J(Y)/\partial U = 0$ implies that $S(Y) = 0$, which, in turn, implies that $\sinh y_{ij} = 0$ and hence $y_{ij} = 0$ for all $i = 1, \cdots, N$ and $j = 1, 2, \cdots, R-1$. Therefore, for $\partial J(Y)/\partial U = 0$ and a fixed B,

$$Y = AU - B = 0$$

which gives a least square fit of

$$U = A^{\#}B. \tag{51}$$

On the other hand, for a fixed U and the constraint $B_j(e_j - e_i) > 0$ for all $i \neq j$ as given in (33), one might attempt to increment B according to the following gradient descent procedure to reduce $J(Y)$ at each step,

$$B(k+1) = B(k) + \delta B(k) \tag{52}$$

where the qth element, $\delta[{}_lb_{jq}(k)]$, of $\delta[{}_lb_j{}^t(k)]$ in $\delta B_j(k)$ is given by

$$\delta[{}_lb_{jq}(k)] = \begin{cases} -p(k)\,{}_l\left[\dfrac{\partial J(Y)(k)}{\partial B}\right]_{jq} \\ \qquad = 2p(k)\ {}_lS_{jq}(Y(k)), \\ \qquad \text{if } {}_lY_j(k)(e_j - e_q) > 0 \\ \qquad \text{for any } q \neq j \\ 0 \qquad \text{if } {}_lY_j(k)(e_j - e_q) \leq 0 \\ \qquad \text{for any } q \neq j. \end{cases}$$

However, ${}_lY_j(k)(e_j - e_q) > 0$ does not imply ${}_lS_j(Y(k))(e_j - e_q) > 0$. In order to make $\delta[{}_lb_j{}^t(k)] \cdot (e_j - e_q) \geq 0$ so that (33) can be satisfied at each step, a modified gradient descent procedure, similar to the one adopted in Teng and Li's generalization of the Ho-Kashyap algorithm,[16] is to be used. Let a $(R-1) \times (R-1)$ non-singular matrix E_j be defined as[16]

$$E_j = [e_j - e_1, \cdots, e_j - e_{j-1}, e_j - e_{j+1}, \cdots, e_j - e_R]. \tag{53}$$

Also define

$$Z_j = Y_jE_j \qquad \text{for all } j = 1, 2, \cdots, R. \tag{54}$$

The increment $\delta[{}_lb_{jq}(k)]$ is then given in terms of

$$\delta[{}_lb^t{}_j(k)E_j]_q = \begin{cases} 2p(k)\ {}_lS_{jq}(Z(k)) \\ \qquad = p(k)[{}_lS_{jq}(Z(k)) + {}_l\Lambda_{jq}(k)] \\ \qquad \text{if } {}_lZ_{jq}(k) \\ \qquad = {}_lY_j(k)(e_j - e_q) > 0 \\ 0 \quad \text{if } {}_lZ_{jq}(k) \\ \qquad = {}_lY_j(k)(e_j - e_q) \leq 0 \end{cases} \tag{55}$$

where

$$ {}_{\iota}\Lambda_{jq}(k) = {}_{\iota}S_{jq}(Z(k))\ \mathrm{Sgn}\ ({}_{\iota}Z_{jq}(k)) \tag{56} $$

and, following (50),

$$ {}_{\iota}S_{jq}(Z(k)) = \mathrm{Sinh}\ {}_{\iota}Z_{jq}(k). \tag{57} $$

Putting into vector representation,

$$ \delta[{}_{\iota}b_j(k)E_j] = p(k)[{}_{\iota}S_j(Z(k)) + {}_{\iota}\Lambda_j(k)] \tag{58} $$

or

$$ \begin{aligned} \delta[{}_{\iota}b_j(k)] &= p(k)[{}_{\iota}S_j(Z(k)) + {}_{\iota}\Lambda_j(k)]E_j^{-1} \\ &= p(k)\ {}_{\iota}H_j(Y(k)) \end{aligned} \tag{59} $$

where

$$ {}_{\iota}H_j(Y(k)) \triangleq [{}_{\iota}S_j(Z(k)) + {}_{\iota}\Lambda_j(k)]E_j^{-1}. \tag{60} $$

$$ H_j(Y(k)) = [S_j(Z(k)) + \Lambda_j(k)]E_j^{-1} $$

$$ H(Y(k)) = \begin{bmatrix} H_1(Y(k)) \\ \cdot \\ \cdot \\ \cdot \\ H_j(Y(k)) \\ \cdot \\ \cdot \\ \cdot \\ H_R(Y(k)) \end{bmatrix} = \begin{bmatrix} {}_{1}H_1(Y(k)) \\ \cdot \\ \cdot \\ \cdot \\ {}_{\iota}H_j(Y(k)) \\ \cdot \\ \cdot \\ \cdot \\ {}_{n_R}H_R(Y(k)) \end{bmatrix} $$

$$ = [h_1(Y(k)) \cdots h_q(Y(k)) \cdots h_{R-1}(Y(k))]. \tag{61} $$

It follows from (58) and (56) that

$$ \delta[{}_{\iota}b_j(k)](e_j - e_i) \geq 0 \qquad \text{for all } i \neq j \quad \text{and for all } j. $$

Then, from (59),

$$ \delta[B(k)] = p(k)H(Y(k)). $$

Substituting the above equation into (52), one has

$$ B(k+1) = B(k) + p(k)H(Y(k)) \tag{62} $$

Using the above equation in (51), one has

$$ \begin{aligned} U(k+1) &= A^{\#}B(k+1) \\ &= A^{\#}\{B(k) + p(k)H[Y(k)]\} \\ &= U(k) + p(k)A^{\#}H[Y(k)] \end{aligned} \tag{63} $$

Therefore, an iterative algorithm to solve for U can be proposed in the following:

$$ \begin{cases} U(0) = A^{\#}B(0) \\ Y(k) = AU(k) - B(k), \qquad Z_j(k) = Y_j(k)E_j \\ B(k+1) = B(k) + p(k)H[Y(k)] \\ H_j(Y(k)) = [S_j(k) + \Lambda_j(k)]E_j^{-1} \\ U(k+1) = U(k) + p(k)A^{\#}H[Y(k)] \end{cases} \tag{64} $$

where $p(k)$ may be chosen as equal to

$$ p(k) = \frac{\sum_{j=1}^{R}\sum_{l=1}^{n_j}\{{}_{\iota}\epsilon_j(k) + {}_{\iota}H_j(Y(k))(E_j^{t})^{-1}R^{-1} \cdot ({}_{\iota}Z_j(k))E_j^{t}{}_{\iota}H_j(Y(k))}{2\sum_{q=1}^{R-1} h_q^{t}(I - AA^{\#})h_q} \tag{65} $$

provided that

$$ \sum_{j=1}^{R}\sum_{l=1}^{n_j}\{{}_{\iota}\epsilon_j(k) + {}_{\iota}H_j(Y(k))(E_j^{t})^{-1}R({}_{\iota}Z_j(k))E_j^{t} \cdot H_j(Y(k))\} > 0 \tag{66} $$

where[17]

$$ R({}_{\iota}Z_j) \triangleq \mathrm{diag}\,[r_{11}({}_{\iota}Z_j), \cdots, r_{R-1,R-1}({}_{\iota}Z_j)] \tag{67} $$

$$ (j = 1, 2, \cdots, R; \qquad l = 1, 2, \cdots, n_j) $$

$$ r_{qq}({}_{\iota}Z_j) \triangleq \frac{\mathrm{Sinh}\ {}_{\iota}Z_{jq}}{{}_{\iota}Z_{jq}} \geq 1, \qquad (q = 1, \cdots, R-1). \tag{68} $$

$$ {}_{\iota}\epsilon_j \triangleq [{}_{\iota}Z_jR({}_{\iota}Z_j) + {}_{\iota}\Lambda_j](E_j^{t}E_j)^{-1}R^{-1}({}_{\iota}Z_j) \cdot [{}_{\iota}Z_jR({}_{\iota}Z_j) - {}_{\iota}\Lambda_j]^{t} \geq 0 \qquad \text{for all } j \quad \text{and all } l. \tag{69} $$

The initial B matrix, $B(0)$, may be chosen from

$$ B^{t}(0) = \beta[e_1, \cdots e_1 \mid \cdots \mid e_j, \cdots, e_j \mid \cdots \mid e_R, \cdots, e_R], \qquad \beta > 0. \tag{70} $$

A recursive relation in $Y(k)$ is also obtained as follows:

$$ Y(k+1) = Y(k) + p(k)(AA^{\#} - I)H[Y(k)] \tag{71} $$

This algorithm is a convergent algorithm for the solution U of the set of linear inequalities (38). The nonlinear separability of the multi-class patterns can also be detected by observing at a certain step k^*

$$ Y_j(k^*)(e_j - e_i) \leq 0 \qquad \text{for all } i \neq j $$

$$ \text{for all } j = 1, 2, \cdots, R. $$

CONVERGENCE PROOF OF THE MULTI-CLASS ALGORITHM

The convergence of the proposed multi-class algorithm can be proved in the following steps.

Lemma 2. Consider the set of inequalities (38) and the

algorithm (64) to solve it. Then

1) $Y_j(k)(e_j - e_i) \gtrdot 0$ for all $i \neq j$
   for all $j = 1, 2, \cdot\cdot, R$
   for any k

2) If (38) is consistent, then
   $Y_j(k)(e_j - e_i) \lessdot 0$ for all $i \neq j$
   for all $j = 1, 2, \cdots, R$
   for any k

This lemma can be proved by contradiction.[17,16]

Theorem II: Consider the set of linear inequalities (38) and the algorithm (64) to solve it, and let

$$V[Y(k)] = \| Y(k) \| \triangleq \mathrm{Tr}\,[Y^t(k)Y(k)] = \sum_{q=1}^{R-1} \| y_q(k) \|^2 = \sum_{j=1}^{R}\sum_{i=1}^{n_j} \| {}_iY_j(k) \|^2 \quad (72)$$

1) If the set of linear inequalities is consistent, then
   a) $\Delta V[Y(k)] \triangleq V[Y(k+1)] - V[Y(k)] < 0$
      and
      $\lim_{k\to 0} V[Y(k)] = 0$ implying convergence to a solution in an infinite number of iterations; and
   b) a solution is obtained in a finite number of steps.
2) If the set of linear inequalities is inconsistent, then there exists a positive integer k^* such that
   $V[Y(k)] < 0$ for $k < k^*$
   $V[Y(k)] = 0$ for $k \geq k^*$
   ${}_iY_j(k)(e_j - e_i) \lessdot 0$ for $k < k^*$
   for all $i \neq j$
   for all $j = 1, 2, \cdots, R$
   ${}_iY_j(k)(e_j - e_i) = Y_j(k^*)(e_j - e_i) \leq 0$
   for all $k \geq k^*$
   for all $i \neq j$
   for all $j = 1, 2, \cdots, R$

and

$U(k) = U(k^*)$ for $k \geq k^*$
$B(k) = B(k^*)$ for $k \geq k^*$.

That is, the occurrence of a matrix $Y(k)$ with all non-positive elements of $Y(k)(e_j - e_i)$ for all $i \neq j$ and all j at any step terminates the algorithm and indicates the nonlinear separability of the R-class patterns.

Proof: Making substitution of the recurrence relation of $Y(k)$ in (71) and simplification, it can be shown that

$$\Delta V[Y(k)] = \mathrm{Tr}\,[Y^t(k+1)Y(k+1) - Y^t(k)Y(k)] = -2p(k)\sum_{j=1}^{R}\sum_{l=1}^{n_j} {}_iH_j(Y(k))\,{}_iY_j{}^t(k) + p^2(k)\sum_{q=1}^{R-1} h_q{}^t(Y(k))(I - AA^{\#})h_q(Y(k)). \quad (73)$$

From (57), (50) and (67),

$${}_iS_j(Z) = {}_iZ_jR({}_iZ_j). \quad (74)$$

Substituting (74) into (60) gives

$${}_iH_j(Y(k)) = [{}_iZ_j(k)R({}_iZ_j(k)) + {}_i\Lambda_j(k)]E_j^{-1}. \quad (75)$$

Substitute (75) and (54) into the following expression,

$$\begin{aligned} &-2p\sum_{j=1}^{R}\sum_{l=1}^{n_j} {}_iH_j\,{}_iY_j{}^t \\ &= -p\sum_j\sum_l {}_iH_j(Y(k))(E_j{}^t)^{-1}R^{-1}({}_iZ_j)E_j{}^t\,{}_iH_j{}^t(Y(k)) \\ &\quad - p\sum_j\sum_l [{}_iZ_jR({}_iZ_j) + {}_i\Lambda_j](E_j{}^tE_j)^{-1}R^{-1}({}_iZ_j) \\ &\quad \cdot [{}_iZ_jR({}_iZ_j) - {}_i\Lambda_j]^t. \end{aligned} \quad (76)$$

It has been shown that the off-diagonal elements in $(E_j{}^tE_j)^{-1}$ are negative,[16] and, from (67) and (68), $R^{-1}({}_iZ_j)$ is a diagonal matrix with all positive diagonal elements. It follows that the off diagonal elements of $(E_j{}^tE_j)^{-1}R^{-1}({}_iZ_j)$ are also negative. From (56), (60), and (74), the elements of $[{}_iZ_jR({}_iZ_j) + {}_i\Lambda_j]$ are either positive or zero, and the corresponding elements of $[{}_iZ_jR({}_iZ_j) - {}_i\Lambda_j]$ are either zero or negative. Hence, the last term in (76), which is equal to $-{}_i\epsilon_j$ as defined in (69), is shown to be non-positive. Substituting (69) into (76), which, in turn, is substituted into (73), one obtains

$$\begin{aligned} \Delta V[Y(k)] = &-p(k)\sum_{j=1}^{R}\sum_{l=1}^{n_j} {}_iH_j(Y(k))\{(E_j{}^t)^{-1} \\ &\cdot [R^{-1}({}_iZ_j) - p(k)I]E_j{}^t\}\,{}_iH_j{}^t(Y(k)) \\ &- p(k)\sum_{j=1}^{R}\sum_{l=1}^{n_j} {}_i\epsilon_j(k) \\ &- p^2(k)\sum_{q=1}^{R-1} h_q{}^t(Y(k))AA^{\#}h_q(Y(k)). \end{aligned} \quad (77)$$

$\Delta V(Y(k))$ is negative definite if the right hand side

of the above equation is negative definite in ${}_lH_j(Y(k))$ or in $[{}_lZ_jR({}_lZ_j) + {}_l\Lambda_j]$. The last two terms on the right hand side are negative semi-definite. If a value of $p(k)$ can be found such that

$$\sum_{j=1}^{R}\sum_{l=1}^{n_j} {}_lH_j(Y(k))\{(E_j^t)^{-1}[R^{-1}({}_lZ_j) - p(k)I]E_j^t\} \cdot {}_lH_j(Y(k)) > 0$$

then $\Delta V(Y(k))$ is negative definite in $[{}_lZ_jR({}_lZ_j) + {}_l\Lambda_j]$. Note that if

$$p(k) = \frac{1}{\cosh Y_{\max}(k)}, \qquad Y_{\max}(k) = \operatorname*{Max}_{j,l,q} | {}_lY_{jq}(k) |,$$

$[R^{-1}({}_lZ_j) - p(k)I]$ is positive definite and has real eigenvalues as can be shown by following (67) and (68); but it is not certain that $(E_j^t)^{-1}[R^{-1}({}_lZ_j) - p(k)I]E_j^t$ can be positive definite for all j and all l. Let $p(k)$ be so chosen as to maximize $-\Delta V[Y(k)]$ at each step, one obtains a choice of $p(k)$ as given in (65), provided the condition (66) is satisfied to make sure that $p(k) > 0$. For this value of $p(k)$,

$$\Delta V(Y(k)) = -\frac{\left[\sum_{j=1}^{R}\sum_{l=1}^{n_j}\{{}_l\epsilon_j(k) + {}_lH_j(Y(k))(E_j^t)^{-1} \cdot R({}_lZ_j(k))E_j^t({}_lH_j(Y(k)))\}\right]^2}{4\sum_{q=1}^{R-1} h_q^t(Y(k))(I - AA^{\#})h_q(Y(k))}$$

≤ 0 for ${}_lH_j(Y(k)) \neq 0$ or $[{}_lZ_jR({}_lZ_j) + {}_l\Lambda_j] \neq 0$ for all l and j.

Hence, $\Delta V[Y(k)]$ is negative definite in $[{}_lZ_jR({}_lZ_j) + {}_l\Lambda_j]$. Note that ${}_lZ_jR({}_lZ_j) + {}_l\Lambda_j = 0$ for all j and all l only if ${}_lZ_j \leq 0$, that is, only if $Y(k) = 0$ or ${}_iY_j(k) \cdot (e_j - e_i) \leq 0$ for all $i \neq j$ and for all j. Since it is assumed that the set of the inequalities (38) is consistent, from the lemma $Y_j(k)(e_j - e_i) \leq 0$ for all $i \neq j$ and for all j; therefore,

$$\Delta V[Y(k)] < 0 \quad \text{for all} \quad Y(k) \neq 0$$
$$= 0 \quad \text{if} \quad Y(k) = 0$$

and the solution $Y = 0$ of the equation (71) can be reached asymptotically, that is,

$$\lim_{k\to\infty} || Y(k) ||^2 = 0$$

which corresponds to a solution U^{**} with $AU^{**} = B$ such that $A_jU^{**}(e_j - e_i) = B_j(e_j - e_i) > 0$ for all $i \neq j$ and for all j. This is the proof of Part 1(a).

Note that for $B(0)$ given in (70), and $\delta > 0$,

$${}_lb^t{}_j(k+1)E_j = {}_lb^t{}_j(k)E_j + p(k)[{}_lS_j(Z(k)) + {}_l\Lambda_j(k)]$$
$$> \beta(1+\delta)e_j^tE_j > 0 \qquad \text{for all} \quad l \quad \text{and} \quad j.$$

For a sufficiently large but finite k, $V[Y(k)] < 1$ such that $|| {}_lY_j(k) ||^2 < 1$ and

$${}_lY_j(k)E_j > -\beta e_j^tE_j \qquad \text{for all} \quad l \neq j \qquad \text{and all} \quad j.$$

It follows then

$$A_jU(k)E_j = B_j(k)E_j + Y_j(k)E_j >$$
$$(1+\delta)B_j(0)E_j - B_j(0)E_j > \delta B_j(0)E_j > 0 \qquad \text{for all} \quad j$$

which indicates a solution $U^* = U(k)$ is obtained in a finite number of iteration steps. This is the proof of part 1(b).

Part 2 can be proved in the same way as that in the Ho-Kashyap theorem.[17]

CONCLUSION

A new generalized inverse algorithm for R-class pattern classification is proposed which is parallel to the one given by Teng and Li. In the case of $R = 2$, the algorithm is reduced to the improved dichotomization algorithm developed in the beginning; except here A_2 is composed of transposes of augmented pattern vectors without change of sign and B_2 is a column vector consisting of elements all equal to $e_2 = -1$. This corresponds to the reformulation of the Ho-Kashyap algorithm as mentioned by Wee and Fu.[15] The proposed 2-class algorithm has a higher rate of convergence than previous methods for a certain range of initial b vector or vectors. A comparison has been made between this improved algorithm with $p(k)$ given by equation (26) and the Ho-Kashyap algorithm with $p = 1$, the convergence rate may be greatly increased for $.001 \leq b_i(0) \leq 0.5$ $(i = 1, 2, \cdots, N)$, as verified by the computer results of several switching theory and pattern classification problems. For problems where a large number of iterations, for example, greater than twenty, were required for the Ho-Kashyap algorithm, the proposed algorithm reduced this number of iterations by a factor of 20 or more. Even though the cost per iteration for the proposed algorithm is 10 to 20 per cent greater than the Ho-Kashyap algorithm, the total cost is reduced. For problems where a small number of iterations were required by the Ho-Kashyap algorithm, less than twenty, the proposed algorithm reduced the number of iterations by as much as 30 percent. Experimental results suggest that the proposed algorithm is advantageous for problems requiring a

large number of iterations by the Ho-Kashyap algorithm.

ACKNOWLEDGMENT

This work is based partially on a Ph.D. dissertation submitted by the first author in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Pittsburgh. During the course of this study, the first author was supported by a NASA Traineeship as well as the Learning Research and Development Center at the University. This work was also supported in part by NASA Grant NsG–416. The authors would like to thank Professor J. G. Castle, Professor T. W. Sze, Dr. T. L. Teng and Major T. E. Brand for their helpful discussions.

REFERENCES

1 N J NILSSON
Learning machines
McGraw–Hill Inc New York N Y 1965

2 G S SEBESTYEN
Decision-making processes in pattern recognition
Macmillan Co New York N Y 1962

3 A G ARKADEV E M BRAVERMAN
Computers and pattern recognition
Thompson Book Co Washington D C 1967

4 L UHR
Pattern recognition
John Wiley and Sons Inc New York N Y 1966

5 C K CHOW
An optimum character recognition system using decision functions
IRE Transactions on Electronic Computers Vol EC-6 No 4 1957

6 R O WINDER
Threshold logic
Ph D Dissertation Princeton University Princeton N J 1962

7 P M LEWIS C L COATES
Threshold logic
John Wiley ane Sons New York N Y 1967

8 Y C HO R L KASHYAP
An algorithm for linear inequalitites and its applications
IEEE Transactions on Electronic Computers Vol EC-14 No 5 1965

9 R L KASHYAP
Pattern classification and switching theory
Ph D dissertation Harvard University Cambridge Mass 1965

10 W G CHAPLIN V S LEVADI
A generalization of the linear threshold decision algorithm to multiple classes
Computer and Information Sciences-II edited by J T Tou Academic Press New York N Y 1967

11 C C BLAYDON
Recursive algorithms for pattern recognition
Ph D dissertation Harvard University Cambridge Mass 1967

12 K S FU W G WEE
On generalizations of adaptive algorithms and application of the fuzzy sets concept to pattern classification
TR-EE67-7 School of Electrical Engineering Purdue University Lafayette Ind 1967

13 I P DEVYATERIKOV A I PROPOI Y Z TSYPKIN
Iterative learning algorithms for pattern recognition
Automation and Remote Control Vol 28 No 1 1967

14 W G WEE
Generalized inverse approach to adaptive multiclass pattern classification
IEEE Transactions on Computers Vol C-17 No 12 1968

15 W G WEE K S FU
An extension of the generalized inverse algorithm to multiclass pattern classification
IEEE Transactions on Systems Science and Cybernetics Vol SSC-4 No 2 1968

16 T L TENG C C LI
On a generalization of the Ho-Kashyap algorithm to multiclass pattern classification
Proceedings of the Third Annual Princeton Conference on Information Sciences and Systems 1969

17 L C GEARY
An improved algorithm for linear inequalities in pattern recognition and switching theory
Ph D Dissertation University of Pittsburgh Pittsburgh Pa 1968

18 A G NIELSEN A H VAGNUCCI C C LI
Application of pattern recognition to electrolyte circadian cycles
Proceedings of the 8th International Conference on Medical and Biological Engineering Chicago Ill 1969

The social impact of computers

by O. E. DIAL

Baruch College
New York, New York

People are afraid of computers,
But they shouldn't be.
Computers are good guys!
MADMAN KRONENBERG

During two recent years at MIT, I had a teletype-writer assigned to my use. The teletype gave me instant access to a large computer complex in which I had various sets of data stored. The teletype was located in a basement room which was generally dark except for the light focused over my teletype and small desk. It was always silent in that room except for the noise of the teletype. In this setting, I would conduct analysis of my data by the hour. I would sort along particular variables, intersect those which seemed promising and in this way be led from one avenue of investigation to another. I was in effect carrying on a dialogue with the computer. I asked a question and I got an answer. The answer led me to other questions. Sometimes the computer would complain that I had not made my inquiry in the correct form and it would suggest that I try again. It kept me informed of the time I had used and how much I had remaining; what data sets I had placed on file and what analysis I had completed. I could cuss it (and often did), thank it, wait impatiently the few seconds it sometimes required to respond, and get excited about what it was telling me. Given all of this, it should not seem strange that this machine came to be human to me for long periods of time. It had personality, value, integrity—and it carried on conversations with me alone. I understand Kronenberg when he says "computers are good guys."[1]

But of course computers are neither good nor bad. They are neutral instruments which are used for good and bad purposes. But their stated purposes often do not take accounts of other effects they are having in society. The purpose of this paper is to speculate on some of those effects.

First let me recount a few statistics which will serve to suggest the magnitude of the subject. After doing so, I believe that you will conclude with me that the most impressive thing about computers is the future, not the past. Over 71,000 computers have been installed in the United States by the time of this writing. Equally important, there are back orders for over 15,000 more.[2] This means that there are back orders in this year alone for more than one-fifth of the total number of computers installed during the past, say, twenty-five years. Demand for computers is changing, becoming stronger.

So much for numbers of computers. How about changes in the computer itself. Paul Armor recently had occasion to report these changes in terms of orders of magnitude.[3] He says that the speed with which computers operate has increased by an order of magnitude about every four years. The size of the computer has decreased an order of magnitude in the last ten years, and it will shrink another three orders of magnitude in the next ten. The cost of computation, too, has declined, by an order of magnitude every four years. To summarize at this point computers are becoming more numerous, faster, smaller and cheaper.

But other changes are taking place. For one, the computer industry is serviced by our system of higher education. In the brief span of years since the term "Computer Scientist" was invented, the system has produced computer scientists at a rate which has already wedged out 2% of America's scientific manpower.[4] At a lower order of preparation, the successive tides of graduates from the countless programming schools in every large city of the country have even now not met demand. Perhaps the best evidence of this unsatisfied demand is the recruiting piracy which has become commonplace in many computer hardware and software companies.

Another area to be noted here is that of computer applications. These have increased along an exponential curve in both breadth and depth. From earlier employment in scientific computation, we now find computers in use by virtually every size and level of public and

private activity, and for an incredible range of applications—from trash-collection routing, to optimization of freight hauling and warehousing; from the operation of traffic control systems, to the optimal stationing of emergency vehicles; from the automation of library inventories and ordering, to the management of military supply and equipment inventories; from hospital patient monitoring, to the conduct of regional health planning; and from matching the unemployed with available jobs, to matching the lonely heart with available dates.

We find similar variety in the users of systems—the clergy, politician, physician, professor, builder, manager, government official, soldier, sportsman, and on to an endless list of persons who just a few years back could not have anticipated their own involvement with computers. For example, a new tide of computer-related technology has made multiple-access networks commonplace, and already we are looking for its marriage to CATV in bringing the blessings of the computer to the housewife. This application will not only require the ultimate in user-oriented languages, but some change in the rules as to who has the last word as well.

In any event, it must be perfectly clear that computers are so well entrenched in every segment of our society that it is merely academic to discuss its impact. There is not much we could do about it at this point even if we tried. But it does make an interesting subject for speculation, which is what I want to try to do at this time.

When a computer is performing in a particular application, its first-order effect is assessed fully in terms of how well it is performing with respect to that application. But taken as a whole, it must be obvious that the computer industry has created a whole range of second and third order effects. Second-order effects might include a wide assortment of contributions, e.g., contribution to GNP, contribution to efficiency in production and administration, contribution to improved scientific and technological capabilities, and a substantial contribution to an improved potential for the collection, storage and selective retrieval of important data and all that this implies.

Third-order effects flow from these contributions. For example, a municipality is for the first time able to create an informational-decision system which is useful for the conduct of its operations, and the planning and evaluation of its programs. This can yield enlightenment in goal formulation and improvements in the quality of life. Whether this potential will be exploited remains to be seen, but it is there, as an indirect effect of computer technology.

But it is not these effects that I am interested in for purposes of this paper. I believe that computer technology has had a number of impacts upon society where the causal relations are even more remote than the examples I have enumerated, and thus more difficult to trace and prove. They must be considered speculative.

First of all, there is a growing persuasion to the systems approach in the belief that it is the only profitable method of inquiry. Of course, there is disenchantment in some quarters, but not enough to slow the movement. This persuasion is understandable given the fact that all computer programs are in fact systems. Input is processed to output. Computer programs are discrete and capable of precise specification. Its processes are clearly visible for inspection and verification. Furthermore, much of it is modular, thus permitting hierarchical structuring as sub-systems into larger systems. The complex can and must become simple with this approach. All mystery is removed. The problem of the social sciences, for example, is merely to isolate and relate the variables in the social system. We can begin at simple levels and build toward the complex. If we are successful, given a variety of inputs for purposes of testing their effects, we can simulate processes within systems at all levels.

Second, I think, is the pervasiveness of programming, or perhaps I should say, the universality of programming. Witness its spread from the computer to becoming a methodology for pedagogy. Considerable attention has been given to the wonders of the programmed textbook, the programmed plan, the programmed career, and so on in a list challenged only by the innovative limits of the entrepreneur. The extent to which programs already govern our thought processes is most appropriate for inquiry. It carries with it a subtle reenforcement of rationality as a value in our society, but rationality as defined in terms of programming. All options are reduced to the program's world of mutually exclusive IF STATEMENTS. Computers and programs are absolutely rational and because of this, they can solve infinitely complex problems with great accuracy, provided that the unravelling can reduce the problem to additions no more complex than a value of one or zero to a value of one or zero. That which cannot be reduced to such algorithms are merely held in abeyance until its true nature can be understood. Understanding is equivalent to order. The reduction of phenomena to specific variables is essential; nothing else will compute. A corollary to this spells the decline of intuition and belief as positive values in society.

Third, I think, developments in computer technology are encouraging streams of reevaluation as to the feasibility of keeping and using historical records of all types. Record reductions may increasingly be based on entire statistical populations, as opposed to sampling.

This can permit, in fact encourage, the collection of environmental and social data on a scale never before contemplated. This may be amassed longitudinally in such quantities and periods as to permit real headway in social sciences research. Such headway has profound implications for the close monitoring of the behavior, activities and welfare of increasingly more numerous segments of society and its institutions. With knowledge and monitoring can come control.

Fourth, I think, the developments I have just discussed will precipitate an increasingly tense confrontation with the individual's right to privacy as a trade-off with society's right to know. To paraphrase Alan Westin, the practical boundaries of privacy, as we knew them before the age of the computer, are being redefined in the onslaught of the greatest data-generating society in human history.[5] Where this will take us is of course unknown, but I deeply suspect that it will be in the direction of acceptance of progressive reductions of the data trails which we now hold to be private. The urgency of the crises presented by over-population and environmental pollution will demand (and we will accede to) planning controls. These are planning and controls which could never have been contemplated without computer technology. The masses of data which are prerequisite would quickly have inundated manual processes of data collection, retrieval and massage. It may well be that privacy is going the way of the skirt length—ever more revealing of the subject it covers.

The remaining areas of impact of computer technology on society seem to me to be relatively trivial, but nonetheless worthy of note. We can anticipate increasingly insistent pressures to articulate the parameters of highly repetitive and routine decisions to the end that they may be automated. This should elevate decision-making in which true judgment is involved to higher orders of application. The lingering worry is, of course, that in our anxiety to do this work we will force the articulation of these parameters, ruling in a measure of cases, however small, in which judgment should remain a factor. This has implications for the demise of concepts of the importance of the individual and of the justice which must be granted him. Where we explicitly settle for validity in two or three sigmas, we are in fact writing off the cases beyond that as unworthy of concern.

While this list of speculations is by no means exhaustive, it should serve to illustrate at least some of the impacts of computer technology upon society. The odd thing is, that we shall not really know what the second and third-order effects are until we have applied computers on a considerable scale to the search. I don't think that this will be done for many years.

The difficulty is, of course, that society does not value information sufficiently at the margin. When computers are employed for public purposes, we demand that their application have a short payoff. Political feasibility is not tested by the automation of personnel accounting systems because savings achieved over manual systems are quickly realized. On the other hand, the development of comprehensive information systems for purposes of collection and storage of environmental and social information have long-run payoffs, and hence do not meet the test of political feasibility. When the time comes to allocate the substantial funds that are required, or to make the organizational, jurisdictional, political and private compromises that are a part of the cost, we effectively reject comprehensive information systems. And yet the problems of our society today are of such a nature that these systems are considerably more important than systems which merely achieve economies of time and dollars. It is quite possible that we should be talking of survival.

I must conclude, therefore, that we will learn of the social impact of computer technology much as we learned of the profound impact of the automotive industry—considerably after the fact.

REFERENCES

1 R TODD
You are an interfacer of black boxes
ATLantic p 68 March 1970

2 *EDP industry report*
pp 6–7 August 6 1969

3 P ARMER
The individual: His privacy self-image and obsolescence
Panel on Science and Technology Eleventh Meeting
Committee on Science and Astronautics U S House of Representatives pp 1–2 January 28 1970

4 *American science manpower*
National Science Foundation NSF 70–5 January 1970

5 A F WESTIN
Privacy and freedom
Atheneum New York 1968

A contiuum of time-sharing scheduling algorithms*

by LEONARD KLEINROCK

University of California
Los Angeles, California

INTRODUCTION

The study of time-sharing scheduling algorithms has now reached a certain maturity. One need merely look at a recent survey by McKinney[1] in which he traces the field from the first published paper in 1964[2] to a succession of many papers during these past six years. Research which is currently taking place within the field is of the nature whereby many of the important theoretical questions will be sufficiently well answered in the very near future so as to question the justification for continuing extensive research much longer without first studying the overall system behavior.

Among the scheduling algorithms which have been studied in the past are included the round robin (RR), the feedback model with N levels (FB_N), and variations of these.[1] The models introduced for these scheduling algorithms gave the designer some freedom in adjusting system performance as a function of service time but did not range over a continuum of system behaviors. In this paper we proceed in that direction by defining a model which allows one to range from the first come first served algorithm all the way through to a round robin scheduling algorithm. We also find a variety of other models within a given family which have yet to be analyzed.

Thus the model analyzed in this paper provides to the designer a degree of freedom whereby he may adjust the relative behavior for jobs as a function of service time; in the past such a parameter was not available. Moreover, the method for providing this adjustment is rather straightforward to implement and is very easily changed by altering a constant within the scheduler.

* This work was supported by the Advanced Research Projects Agency of the Department of Defense (DAHC15-69-C-0285).

A GENERALIZED MODEL

In an earlier paper[3] we analyzed a priority queueing system in which an entering customer from a particular priority group was assigned a zero value for priority but then began to increase in priority linearly with time at a rate indicative of his priority group. Such a model may be used for describing a large class of time-sharing scheduling algorithms. Consider Figure 1. This figure defines the class of scheduling algorithms which we shall consider. The principle behind this class of algorithms is that when a customer is in the system waiting for service then his priority (a numerical function) increases from zero (upon his entry) at a rate α; similarly, when he is in service (typically with other customers sharing the service facility simultaneously with him as in a processor shared system[4]) his priority changes at a rate β. All customers possess the same parameters α and β. Figure 1 shows the case where both α and β are positive although, as we shall see below, this need not be the case in general. The history of a customer's priority value then would typically be as shown in Figure 1 where he enters the system at time t_0 with a 0 value of priority and begins to gain priority at a rate α. At time t_1 he joins those in service after having reached a value of priority equal to $\alpha(t_1 - t_0)$. When he joins those in service he shares on an equal basis the capacity of the service facility and then continues to gain priority at a different rate, β. It may be that a customer is removed from service before his requirement is filled (as may occur when one of the slopes is negative); in this case, his priority then grows at a rate of α again, etc. At all times, the server serves *all* those with the highest value of priority. Thus we can define a slope for priority while a customer is queueing and another slope for priority while a cus-

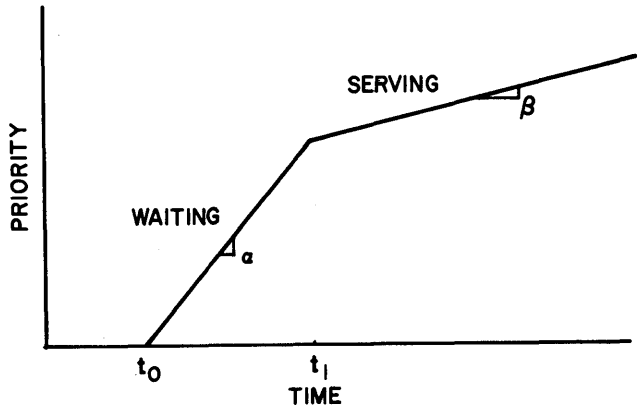


Figure 1—Behavior of the time-varying priority

tomer is being served as

$$\text{queueing slope} = \alpha \tag{1}$$

$$\text{serving slope} = \beta. \tag{2}$$

A variety of different kinds of scheduling algorithms follow from this model depending upon the relative values of α and β. For example, when both α and β are positive and when $\beta \geq \alpha$ then it is clear that customers in the queue can never catch up to the customer in service since he is escaping from the queueing customers at least as fast as they are catching up to him; only when the customer in service departs from service after his completion will another customer be taken into service. This new customer to be taken into the service facility is that one which has the highest value of priority. Thus we see that for the range

$$0 < \alpha \leq \beta \tag{3}$$

we have a pure *first come first served* (FCFS) scheduling algorithm. This is indicated in Figure 2 where we show the entire structure of the general model.

Now consider the case in which

$$0 \leq \beta \leq \alpha. \tag{4}$$

This is the case depicted in Figure 1. Here we see that the group of customers being served (which act among themselves in a processor-shared round robin (RR) fashion) is attempting to escape from the group of customers in the queue; their attempt is futile, however, and it is clear from this range of parameters that the queueing customers will eventually each catch up with the group being served. Thus the group being served is selfishly attempting to maintain the service capacity for themselves alone and for this reason we refer to this system as the *selfish round robin* (SRR).

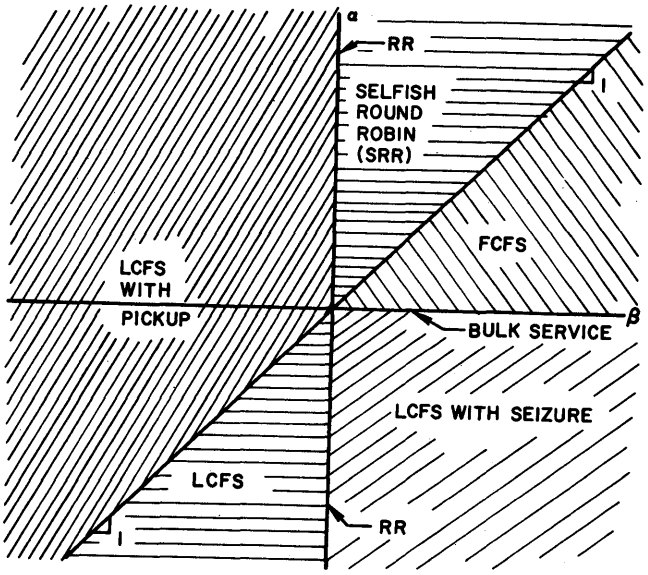


Figure 2—The structure of the general model

What happens in this case is that entering customers spend a period of time in the queue and after catching up with the serving group proceed to be served in a round robin fashion. The duration of the time they spend in the queue depends upon the relative parameters α and β as we shall see below. It is clear however that for $\beta = 0$ we have the case that customers in service gain no priority at all. Thus any newly entering customer would have a value of priority exactly equal to that of the group in service and so will immediately pass into the service group. Since all serving customers share equally, we see that the limiting case, $\beta = 0$, is a processor-sharing round robin (RR) scheduling algorithm! It happens that SRR yields to analysis very nicely (whereas some of the other systems mentioned below are as yet unsolved) and the results of this analysis are given in the next section.

Another interesting range to consider is that for which

$$\alpha \leq \beta < 0. \tag{5}$$

Here we have the situation in which queueing customers lose priority faster than serving customers do; in both cases however, priority decreases with time and so any newly entering customer will clearly have the highest priority and will take over the complete service facility for themselves. This most recent customer will continue to occupy the service facility until either he leaves due to a service completion or some new customer enters the system and ejects him. Clearly what we have here is a classical *last come first served* (LCFS) scheduling algorithm as is indicated in Figure 2.

Now consider the range

$$\alpha < 0 < \beta. \tag{6}$$

In this case a waiting customer loses priority whereas a customer in service gains priority. When an arriving customer finds a customer in service who has a negative value for priority then this new customer preempts the old customer and begins service while at the same time his priority proceeds to increase at a rate β; from here on no other customer can catch him and this customer will be served until completion. Upon his completion, service will then revert back to that customer with the largest value of priority. Since customers lose priority with queueing time, then all customers in the system when our lucky customer departed must have negative priority. One of these will be chosen and will begin to gain priority; if now he is lucky enough to achieve a positive priority during his service time, then he will seize the service facility and maintain possession until his completion. Thus we call this range *LCFS with seizure* (see Figure 2).

In the special case

$$\alpha = 0 < \beta \tag{7}$$

we have the situation in which a newly emptied service facility will find a collection of customers who have been waiting for service and who have been kept at a zero value priority. Since all of these have equal priority they will all be taken into service simultaneously and then will begin to gain priority at a rate $\beta > 0$. Any customers arriving thereafter must now queue in bulk fashion since they cannot catch up with the current group in service. Only when that group finishes service completely will the newly waiting group be taken into service. We refer to this case as *bulk service.*

The last case to consider is in the range

$$\beta < 0, \qquad \beta < \alpha. \tag{8}$$

In this case a customer being served always loses priority whereas a queueing customer loses priority at a slower rate or may in fact gain priority. Consequently, serving customers will tend to "run into" queueing customers and pick them up into the service facility at which point the entire group continues to decrease in priority at rate β. We refer to this region as *LCFS with pickup* (see Figure 2).

Thus Figure 2 summarizes the range of scheduling algorithms which this two-parameter priority function can provide for us. We have described a number of regions of interest for this class of algorithms. The FCFS, LCFS, and RR systems, of course, are well known and solved. The three regions given by Equations 4, 6, and 8 are as yet unsolved. As mentioned before, the SRR system yields very nicely to analysis and that analysis is given in this paper. This system has the interesting property that we may vary its parameters and pass smoothly from the FCFS system through the SRR class to the familiar RR system. The others (LCFS with seizure and LCFS with pickup) are as yet unsolved and appear to be more difficult to solve than the SRR. Of course other generalizations to this scheme are possible, but these too are yet to be studied. Among these generalizations, for example, is the case where each customer need not have the same α and β; also one might consider the case where growth (or decay) of priority is a non-linear function of time. Of all these cases we repeat again that the SRR has been the simplest to study and its analysis follows in the next section.

THE SELFISH ROUND ROBIN (SRR) SCHEDULING ALGORITHM

We consider the system for which customers in service gain priority at a rate less than or equal to the rate at which they gained priority while queueing (see Equation (4)); in both cases the rate of gain is positive. We assume that the arrival process is Poisson at an average rate of λ customers per second

$$P[\text{inter-arrival time} \leq t] = 1 - e^{-\lambda t} \quad t \geq 0 \tag{9}$$

and that the service times are exponentially distributed

$$P[\text{service time} \leq x] = 1 - e^{-\mu x} \quad x \geq 0 \tag{10}$$

Thus the two additional parameters of our system are

$$\text{average arrival rate} = \lambda \tag{11}$$

$$\text{average service time} = 1/\mu \tag{12}$$

As usual, we define the utilization factor

$$\rho \equiv \lambda/\mu \tag{13}$$

For the range of α, β under consideration it is clear that once a customer enters the service facility he will not leave until his service is complete. Consequently, we may consider the system as broken into two parts: first, a collection of queued customers; and second, a collection of customers in service. Figure 3 depicts this situation where we define*

$$T_w = E[\text{time spent in queue box}] \tag{14}$$

$$T_s = E[\text{time spent in service box}] \tag{15}$$

$$N_w = E[\text{number in queue box}] \tag{16}$$

$$N_s = E[\text{number in service box}] \tag{17}$$

* The notation $E[x]$ reads as "the expectation of x."

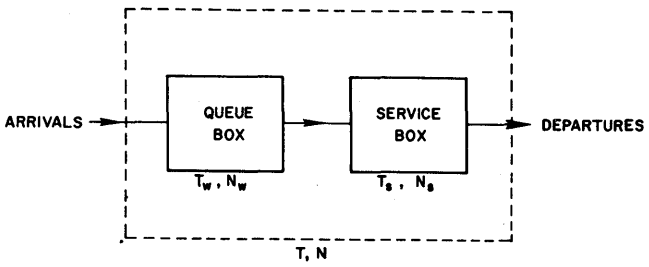


Figure 3—Decomposition of the SRR system

We further define

$$T = T_w + T_s = E[\text{time in system}] \quad (18)$$

$$N = N_w + N_s = E[\text{number in system}] \quad (19)$$

Due to the memoryless property of the exponential service time distribution, it is clear that the average number in system and average time in system are independent of the order of service of customers; this follows both from intuition and from the conservation law given in Reference 5. Thus we have immediately

$$T = \frac{1/\mu}{1 - \rho} \quad (20)$$

$$N = \frac{\rho}{1 - \rho} \quad (21)$$

For our purposes we are interested in solving for the average response time for a customer requiring t seconds of service; that is for a customer requiring t seconds of complete attention by the server or $2t$ seconds of service from the server when he is shared between two customers, etc. Recall that more than one customer may simultaneously be sharing the attention of the service facility and this is just another class of processor-sharing systems.[4] Thus our goal is to solve for

$$T(t) = E[\text{response time for customer requiring } t \text{ seconds of service}] \quad (22)$$

where by response time we mean total time spent in the system. The average of this conditional response time without regard to service time requirement is given by Equation 20. Due to our decomposition we can write immediately

$$T(t) = T_w(t) + T_s(t) \quad (23)$$

where $T_w(t)$ is the expected time spent in the queue box for customers requiring t seconds of service and $T_s(t)$ is the expected time spent in the service box for customers requiring t seconds of service. Since the system is unaware of the customer's service time until he departs from the system, it is clear that the time he spends in the queue box must be independent of this service time and therefore

$$T_w(t) = T_w \quad (24)$$

Let us now solve for $T_s(t)$. We make this calculation by following a customer, whom we shall refer to as the "tagged" customer, through the system given that this customer requires t seconds of service. His time in the queue box will be given by Equation 24. We now assume that this tagged customer has just entered the service box and we wish to calculate the expected time he spends there. This calculation may be made by appealing to an earlier result. In Reference 4, we studied the case of the processor-shared round robin system (both with and without priorities). Theorem 4 of that paper gives the expected response time conditioned on service time and we may use that result here since the system we are considering, the service box, appears like a round robin system. However, the arrival rate of customers to the service box conditioned on the presence of a tagged customer in that box is no longer λ, but rather some new average arrival rate λ'. In order to calculate λ' we refer the reader to Figure 4. In this figure we show that two successive customers arrive at times t_1 and t_2 where the average time between these arrivals is clearly $1/\lambda$. The service group moves away from the new arrivals at a rate β and the new arrivals chase the service group at a rate α; as shown in Figure 4, these two adjacent arrivals catch up with the service group where the time between their arrival to the service box is given by $1/\lambda'$. Recall that the calculation we are making is conditioned on the fact that our tagged customer remains in the service box during the interval of interest; therefore the service box is guaranteed not to empty over the period of our

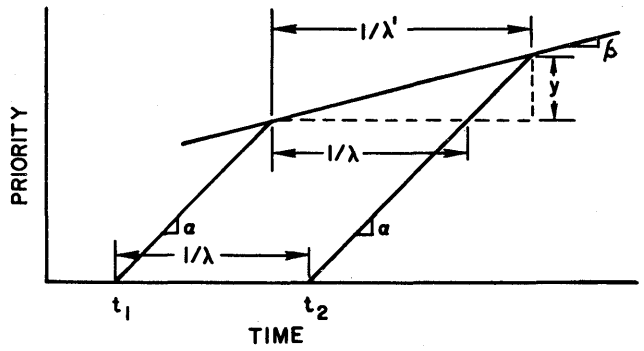


Figure 4—Calculation of the conditional arrival rate to the service box

calculations. λ' is easily calculated by recognizing that the vertical offset y may be written in the following two ways

$$y = \left(\frac{1}{\lambda'}\right)\beta$$

$$y = \left(\frac{1}{\lambda'} - \frac{1}{\lambda}\right)\alpha$$

and so we may solve for λ' as follows

$$\lambda' = \lambda\left(1 - \frac{\beta}{\alpha}\right) \tag{25}$$

(recall that for the SRR system $\beta \leq \alpha$). For convenience we now define

$$\rho' = \lambda'/\mu \tag{26}$$

We may now apply Theorem 4 of Reference 4 and obtain the quantity we are seeking, namely,

$$T_s(t) = \frac{t}{1 - \rho'} \tag{27}$$

The only difference between Equation 27 and the referenced theorem is that here we use ρ' instead of ρ since in all cases we must use the appropriate utilization factor for the system under consideration. That theorem also gives us immediately that

$$N_s = \frac{\rho'}{1 - \rho'} \tag{28}$$

This last equation could be derived from Equation 27 and the application of Little's result[6] which states that

$$\lambda' T_s = N_s \tag{29}$$

and where

$$T_s \equiv \int_0^\infty T_s(t)\mu e^{-\mu t}\, dt$$

$$T_s = \frac{1/\mu}{1 - \rho'} \tag{30}$$

We may now substitute Equation 27 into Equation 23 to give

$$T(t) = T_w + \frac{t}{1 - \rho'} \tag{31}$$

In order to evaluate T_w we form the average with respect to t over both sides of Equation 31 to obtain

$$\int_0^\infty T(t)\mu e^{-\mu t}\, dt = T_w + \int_0^\infty \frac{t}{1 - \rho'}\mu e^{-\mu t}\, dt$$

and so

$$T = T_w + \frac{1/\mu}{1 - \rho'} \tag{32}$$

Using Equation 20 we have the result

$$T_w = \frac{1/\mu}{1 - \rho} - \frac{1/\mu}{1 - \rho'} \tag{33}$$

Upon substituting Equation 33 into Equation 31 we obtain our final result as

$$T(t) = \frac{1/\mu}{1 - \rho} + \frac{t - 1/\mu}{1 - \rho'} \tag{34}$$

Another convenient form in which to express this result is to consider the average time wasted in this SRR system where wasted time is any extra time a customer spends in the system due to the fact that he is sharing the system with other customers. Thus, by definition, we have

$$W(t) = T(t) - t \tag{35}$$

and this results in

$$W(t) = \frac{\rho/\mu}{1 - \rho} + \frac{(t - 1/\mu)\rho'}{1 - \rho'} \tag{36}$$

In both Equations 34 and 36 we observe for the case of a customer whose service time is equal to the average service time $(1/\mu)$ that his average response time and average wasted time are the same that he would encounter for any SRR system; thus his performance is the same that he would receive, for example, in a FCFS system. We had observed that correspondence between the RR system and the FCFS system in the past; here we show that it holds for the entire class of SRR systems. In Figure 5 below we plot the performance of the class of SRR systems by showing the dependence of the wasted time for a customer whose service time is t seconds as a function of his service time. We show this for the case $\rho = \frac{3}{4}$ and $\mu = 1$. The truly significant part regarding the behavior of the SRR system is that the dependence of the conditional response time upon the service time is linear. Once observed, this result is intuitively pleasing if we refer back to Figure 3. Clearly, the time spent in the queue box is some constant independent of service time. However, the time spent in the service box is time spent in a round robin system since all customers in that box share equally the capability of the server; we know that the response time for the round robin system is directly proportional to service time required (in fact, as shown in Reference 8, this statement is true even for arbitrary service time). Thus the total time spent in the SRR system must be equal to some con-

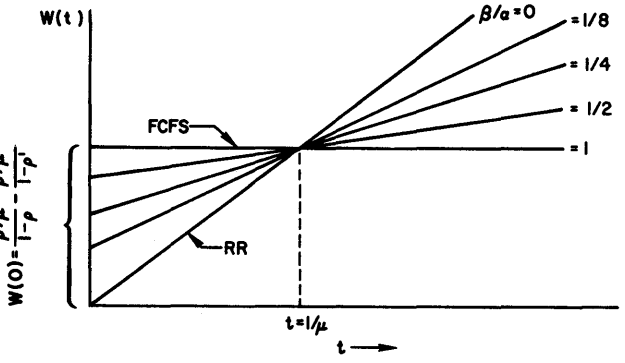


Figure 5—Performance of the SRR system

stant plus a second term proportional to service time as in fact our result in Equation 34 indicates. Again we emphasize the fact that customers whose service time requirements are greater than the average service time requirement are discriminated against in the SRR system as compared to a FCFS system; conversely, customers whose service time requirement is less than the average are treated preferentially in the SRR system and compared to the FCFS system. The degree of this preferential treatment is controlled by the parameters α and β giving the performance shown in Figure 5.

CONCLUSION

In this paper we have defined a continuum of scheduling algorithms for time-shared systems by the introduction of two new parameters, α and β. The class so defined is rather broad and its range is shown in Figure 2. We have presented the analysis for the range of parameters that is given in Equation 4 and refer to this new system as the selfish round robin (SRR) scheduling algorithm. Equation 34 gives our result for the average response time conditioned on the required service time and we observed that this result took the especially simple form of a constant plus a term linearly dependent upon the service time. Moreover, we observe that the parameters α and β appear in the solution only as the ratio β/α. This last is not overly surprising since a similar observation was made in the paper[3] which was our point of departure for the model described herein; namely, there too the slope parameters appeared only as ratios. Thus in effect we have introduced one additional parameter, the ratio β/α, and it is through the use of this parameter that the designer of a time-sharing scheduling algorithm is provided a degree of freedom for adjusting the extent of discrimination based upon service time requirements which he wishes to introduce into his algorithm; the implementation of this degree of freedom is especially simple. The range of the algorithm is from the case where there is zero discrimination based on service time, namely the FCFS system, to a case where there is a strong degree of discrimination, namely the RR system.

The mathematical simplicity of the SRR algorithm is especially appealing. Nevertheless, the unsolved systems referred to in this paper should be analyzed since they provide behavior distinct from the SRR. In any event, this continuum of algorithms is simply implemented in terms of the linear parameters α and β, and the scheduling algorithm can easily choose the desired behavior by adjusting α and β appropriately.

REFERENCES

1 J M MCKINNEY
A survey of analytical time-sharing models
Computing Surveys Vol 1 No 2 pp 105–116 June 1969

2 L KLEINROCK
Analysis of a time-shared processor
Naval Research Logistics Quarterly Vol 11 No 1 pp 59–73 March 1964

3 L KLEINROCK
A delay dependent queue discipline
Naval Research Logistics Quarterly Vol 11 No 4 pp 329–341 December 1964

4 L KLEINROCK
Time-shared systems: A theoretical treatment
JACM Vol 14 No 2 pp 242–261 April 1967

5 L KLEINROCK
A conservation law for a wide class of queueing disciplines
Naval Research Logistics Quarterly Vol 12 No 2 pp 181–192 June 1965

6 J D C LITTLE
A proof of the queueing formula $L = \lambda W$
Operations Research Vol 9 pp 383–387 1961

7 L KLEINROCK
Distribution of attained service in time-shared systems
J of Computers and Systems Science Vol 3 pp 287–298 October 1967

8 M SAKATA S NOGUCHI J OIZUMI
Analysis of a processor shared queueing model for time-sharing systems
Proc of the Second Hawaii International Conference on Systems Science pp 625–628
University of Hawaii Honolulu Hawaii January 22–24 1969

The management of a multi-level non-paged memory system

by FOREST BASKETT, J. C. BROWNE and WILLIAM M. RAIKE

The University of Texas
Austin, Texas

INTRODUCTION

There is a clear tendency for large-scale and, especially time-sharing computer systems to have several levels of random access memory with gradations in access time, degree of addressability, and functional capability. In our configuration at The University of Texas at Austin these are a high-speed magnetic core memory, an extended core memory of magnitude 4 times the size of the main memory, and 4 large, fast disks. An extensive literature[1,2,3,4] has already developed on the management of multi-level systems where the main memory is structured in pages, usually with an extended logical addressing space.

The management of multi-level memory systems where the main memory is not paged has received much less attention.[5,6,7] Certain problems are characteristic of systems which can assign main memory to a given process only in a single contiguous block. These problems become performance-limiting factors when the computer system supports a multi-programming batch system and a substantial interactive load. We discuss some models for memory management where both a multiprogramming batch system and heavy interactive usage compete for the resources of a three-level memory system with a non-paged main memory. These models are based on detailed measurements[8] of system performance and job characteristics for the current operation of a CDC 6600 computer system. We pay special attention to the competition between batch and interactive jobs for memory resources and on the costs of data flow between levels of random access memory.

Fuchel and Heller[5] have studied the general characteristics of Control Data's extended core storage (ECS) as a swapping medium and a storage medium for active files. Fuchel, Campbell, and Heller[15] have studied in detail the use of ECS as a buffering device for active files with the motive of increasing CPU efficiency. Such uses of ECS in our configuration does little to improve CPU efficiency. As predicted by their analysis, the use of two dual-channel 6638 disks (effectively, four independent disks), together with 131,000 words of main memory, allows us to attain central processor efficiency in the range 85 to 90 percent.

Our intention is to use ECS to provide a significant interactive computing capability without significantly degrading the current high level of system performance. The following analysis will show how that is possible.

SYSTEM CHARACTERISTICS

The particular system which we use to parameterize our models is a CDC 6600 with 131,000 60-bit words of high-speed main memory, 524,000 words of extended core storage (ECS), and four 6-million-word disks.

For future reference, we shall summarize certain characteristics of the CDC ECS: (a) Transfers between ECS and main memory proceed at the acceptance rate of main memory (10 words per microsecond) after a transfer is initiated. (b) Average transfer initiation time is 3.4 microseconds. (c) Transfers are initiated by central processor instructions and hold the central processor until the transfer is complete. (d) ECS is internally structured in 8-word records. A peripheral processor may interrupt an ECS transfer at the end of an 8-word record. One main memory word may be read or written in one microsecond by the peripheral processor before the transfer is automatically resumed with an additional start-up time. Each main memory access by a peripheral processor during an ECS transfer will be delayed an average of 400 nanoseconds and will increase the total transfer time by 4.4 microseconds. (e) ECS is word addressable by the central processor for data transfers between it and main memory, but instructions and data cannot be fetched directly to CPU registers. Thus ECS cannot be used as a direct logical extension of main memory, as IBM Large Core Storage can, but must be considered as auxiliary storage.

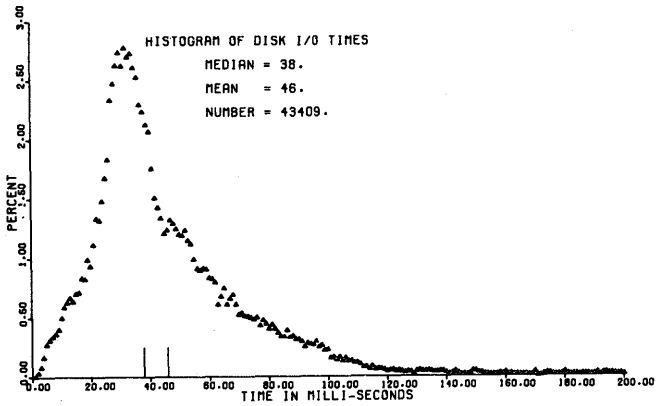


Figure 1—Histogram of disk I/O times

The operating system for The University of Texas at Austin 6600 system supports both local site and remote batch job entry as well as conversational interaction. The operating system is locally written, having been derived from an early (1966) version of the standard CDC (SCOPE 2.0) operating system. We shall describe those features which are essential to the management of memory resources.

The operating system divides main memory into not more than eight blocks or work spaces. One work space is a fixed length block for system tables and monitor code. Up to seven variable length work spaces may be allocated to active processes. Each assigned work space is associated with a control point. One control point is used to control and drive the peripheral input and output equipment and the remote computers. Another is assigned to the management of the remote on-line terminals. Five control points and 85,000 words of main memory are left for user programs. Administrative policy constrains batch jobs to 73,000 words or less and interactive jobs to 32,000 words or less.

Files are assigned to the four disks in a round-robin fashion. The next file to be assigned is assigned to the next disk in the round-robin. No space is allocated on that disk until the file is actually written. The disks have moveable arms but only 32 different positions. The basic allocation unit on a disk is called a half-track, 3072 words, comprising every other sector (64 words) of a particular arm position and head select. Each arm position covers 64 half-tracks. Half-tracks are numbered according to their physical order on the disk. At the moment in time when a file needs more space, the lowest numbered half-track available on the disk to which the file is assigned is allocated to that file. The effect of these facts and allocation policies is that currently active files are distributed over the available disks and active files that are on the same disk tend to be interleaved under a given arm position. This minimizes the principal difficulty with moveable arm disks, namely arm motion. Figure 1 is a histogram of user disk I/O times under these policies.

The relevant data on the 6638 disks with respect to this figure is that the average rotational latency is 25 milliseconds and arm motion requires between 20 and 100 milliseconds, depending on the distance involved.

For practical purposes, we take this structure as given and we consider next the allocation and scheduling strategies within this structure which affect system performance.

ALLOCATION OF RESOURCES BETWEEN BATCH AND INTERACTIVE USAGE

It is desired to allocate the resources of the system; access to the CPU, main memory, ECS, etc. so as to insure rapid (one second or less) response time to a substantial number (e.g., about 30–40) of interactive users while maintaining a fast batch system throughput and a high ($\geqq 80\%$) central processor efficiency. The primary factors in determining the allocation strategy are job load characteristics, the characteristics and capacity of the swapping media, the swapping overhead, and the competition between the batch and interactive job streams. These problems and the allocation of CPU activity between control points are discussed in the next sections. The interference between the batch and interactive systems is primarily a competition for main memory. The factors dominating this competition are interference with disk I/O due to main memory lockout during swapping, the scheduling of batch jobs for loading into main memory, and the "memory compacting" problem. A separate section is devoted to each of these factors.

JOB-LOAD CHARACTERISTICS

The job-load characteristics have been determined by measurement of more than 50,000 jobs. The measured mean size of user batch programs is 21,000 words. The mean interactive program size is 12,800 words.

Let S denote the size on an arbitrarily chosen active job and $F_S(\beta) = P[S \leqq \beta]$ be the distribution function of job sizes. Figure 2 shows $F_S(\beta)$ for batch and interactive jobs. At any given time, the probability that any given batch job in execution will be 21,000 words or less in size is one-half. The core-size distribution is such that five control points are active 20 percent of the time and four are active 65 percent of the time.

The mean I/O wait time (t_{IO}) is 46 milliseconds. The distribution function of t_{IO} is extremely compact as can be seen from Figure 1. Disk channels are active 22 percent of the time per channel. This means that a disk I/O request is queued because of conflicting channel usage only a small percentage of the time. The compactness of this distribution is produced by the disk space allocation strategies described in the previous section. The mean time (t_c) a batch job computes before requesting an I/O operation is 48 milliseconds. The mean CPU time per interaction is not much above one millisecond. The measured median "think time" for an interactive user is 10 seconds (12). This figure is very close to that found in studies of other systems.[13,14]

THE ALLOCATION OF THE CPU TO ACTIVE JOBS

Modeling and analysis done by Gaver[17] has shown that there are four principal effects on CPU efficiency in a multiprogramming computer system: (1) the ratio of t_c, the average interval of execution between I/O operations for any single given job and t_{IO}, the average length of an I/O operation; (2) J, the degree of multiprogramming or the number of jobs in main memory executing, doing I/O, or awaiting either; (3) I, the number of I/O units; and (4) σ^2, the variance of the compute time distribution. It is interesting to note that the "shape" of the compute time distribution curve seems relatively insignificant compared to the value of σ^2.

In order to increase the degree of multiprogramming, we make use of "memory compacting." The operating system will *move* the contiguous block of memory assigned to a control point from one absolute location to another. This increase in the degree of multiprogramming is achieved by the job scheduling policy developed in a following section. The cost of memory compacting under the resulting policy depends on the packing policy used. The last section describes a policy which makes this cost very small (less than one percent of the CPU time) with respect to the increased CPU efficiency gained from the increased degree of multiprogramming, even with a heavy interactive job load. In actual practice we observe a CPU efficiency between 85 and 90 percent.

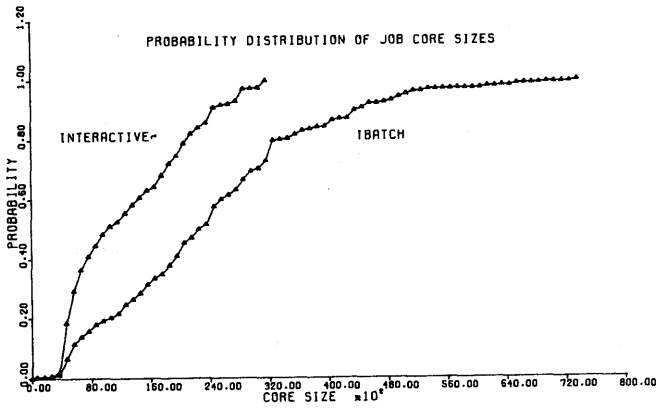


Figure 2—Probability distribution of job core sizes

The variance, σ^2, of the CPU compute time distribution can be affected by CPU scheduling strategies. Gaver's analysis[17] considered no CPU scheduling other than first come, first served. However, by switching the CPU among jobs ready to compute, we can effectively lower the variance of the compute time distribution. Lowering σ^2 will increase the CPU efficiency as well as increase the I/O rate. A full analysis of this effect has not yet been completed but preliminary results on a round-robin servicing discipline indicate that there is an optimum quantum size for a given distribution of I/O operations and compute times and a given cost of switching the CPU from one job to another. This optimum is most affected by the cost of switching. For a CDC 6600, this cost is approximately 32 microseconds. A quantum size of five milliseconds seems to achieve the best results with respect to increasing the I/O rate *and* increasing the CPU efficiency.

ALLOCATION OF ECS

Despite the speed of the disks with respect to the I/O demands of the batch system, it is easy to see that they are extremely slow swapping devices for a non-paged system, especially compared to ECS. For the mean interactive job size of 12,800 words a disk transfer would require at least 250 milliseconds exclusive of any queueing and positioning time. The transfer time for ECS depends on the block size used. Figure 3 shows the actual transfer time as a percentage of the maximum theoretical transfer time as a function of the block size. It should be noted that the limiting percentage of 64 percent is due to the PP break-ins experienced in a running system. This curve makes clear the desirability of a large block size to take maximum advantage of the speed of ECS. Thus we want B, the block size to be as large as possible. However, if $\bar{S}$ is the mean program size and $B \ll \bar{S}$ then the waste per program is $B/2$. We now require that the waste, $B/2/\bar{S}$, be less than 2 percent. $B = 512$ achieves this objective while giving an ECS transfer utilization rate near 50 percent. Since $B \ll S$ for this value, the result is valid.

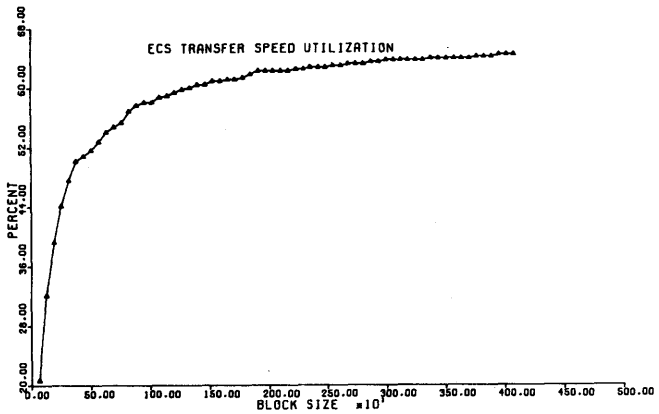


Figure 3—ECS transfer speed utilization

Because of the unsuitability of the disks for interactive swapping, the memory capacity of ECS effectively defines the "natural capacity" of the system for handling interactive jobs. The interactive job-size distribution shows the ECS capacity to be about 42 users. Recalling that measured median think time for a given interactive user is 10 seconds and the mean CPU time per interaction is on the order of one millisecond, it is clear that with the CPU scheduling policy described above, a single control point could service interactive demands up to the "natural capacity" of the system with response times of less than one second. A CPU quantum size of five milliseconds insures CPU allocation to the interactive control point at least once every 25 milliseconds so that there is no appreciable delay due to CPU queueing. The interaction rate of once every 250 milliseconds for a set of 40 interactive users and a mean swap time (in and out) of 5.2 milliseconds will give a CPU overhead in this case of approximately 2 percent. We consider this to be a very reasonable price to pay for the attained interactive service.

INTERFERENCE WITH DISK I/O DUE TO SWAPPING

The most serious effect of swapping interactive jobs through ECS is the effect on disk I/O. The alternate sector coding technique used on the disks requires that the peripheral processors doing disk I/O have sufficient access to main memory between alternate sectors to access 64 words. The transfer itself between main memory and peripheral processor memory requires 320 microseconds. The total available time between alternate sectors is 500 microseconds. Because of the bookkeeping to be done, there is very little time to spare. Delays in accessing main memory will cause the peripheral processors to miss the next sector and have to wait a full revolution of the disk to access the missed sector. Since ECS transfers with a block size as small as 8 words cause sufficient delay (in main memory access for the peripheral processors) to generate this problem, it is important to assess the level of ECS usage and the amount of this disk I/O degradation. A missed sector adds 50 milliseconds to the I/O service time for an I/O request that encounters this problem. The level of ECS usage predicted in the preceding analysis will cause an increase of approximately 15 percent in the average I/O service time. This will cause an increase in the probability that a process is in an I/O wait state and thus an increase in the expected CPU idle time. This effect may require a change in the alternate sector coding technique used on the disks. On CDC 6638 disks, without ECS interference, the alternate sector coding technique is so close to optimal that a high level of ECS usage could be counterproductive; however, preliminary studies indicate a decrease in expected CPU efficiency of about 5 or 6 percent at peak periods. Thus the total expected overhead and increased CPU idle time is less than 10 percent, a figure we consider acceptable.

BATCH JOB SCHEDULING

To schedule jobs, the basic administrative policy is to give fast turnaround to jobs with small resource requirements. We can justify this policy on the basis of job load. For example, 90 percent of all jobs use less than 20 percent of all CPU time charged to users.

The scheduler has available the same swapping mechanism for batch jobs as for interactive jobs. A scheduling strategy which pre-empts a long batch job when a short job arrives in the queue and resumes the long job after the short job terminates can be used. This is commonly called a pre-emptive-resume type of scheduling strategy. We desire to find a scheduling strategy which makes "optimal" use of the available space with respect to the above policy.

To describe the situation formally, we make use of a simple type of mathematical programming model. The scheduling problem consists, essentially, of examining the n jobs which are currently awaiting execution and selecting some or all of these for loading. Let x_i be a variable assuming the values 0 or 1, denoting respectively the decision not to load, or to load, job i. Each job i has a space requirement s_i, and at scheduling time has a time remaining requirement t_i, the total time requirement for job i minus the elapsed execution

time. If the main memory available for batch jobs is S, and if each job is assumed to have a "utility" $u_i = 1/t_i$ (which represents the anticipated completion rate for job i), one way of formulating the scheduling problem is to determine values for the variables $x_1, \cdots, x_n$ which solve

$$\text{Maximize } \sum_{i=1}^{n} u_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} s_i x_i \leqq S \tag{1}$$

$$\text{and } x_i = 0 \text{ or } 1.$$

Since u_i and s_i are strictly positive for all i, this is the familiar "knapsack problem" of mathematical programming. The actual scheduling algorithm will *not* be determined by solving (1) for several reasons. First, the available space S is not constant over the time interval within which a given scheduling decision will be effective and thus is not known with certainty. This is a result of the rapidly changing space requirements of the control point assigned to interactive jobs. Second, the discrete character of the variables x_i makes an optimal schedule determined by (1) unduly sensitive to fluctuations in S, an undesirable lack of robustness of the solution. For this reason, we reformulate (1) to take into account the uncertain nature of S. Specifically, we suppose S is a random variable with known distribution function F_S. This function can be determined from the data presented graphically in Figure 1. In view of the "policy" character (versus "resource" character) of the space constraint, a particularly appealing way to encompass the intent of (1) is to express the space restriction as a "chance constraint."[16] We rewrite (1) as follows

$$\text{Maximize } \sum_{i=1}^{n} u_i x_i$$

$$\text{subject to } P\left[\sum_{i=1}^{n} s_i x_i \leqq S\right] \geqq \alpha \tag{2}$$

$$\text{and } 0 \leqq x_i \leqq 1.$$

Here α is the confidence with which we require the space constraint to be satisfied. A typical value for α, which is specified *a priori* and is *not* a variable whose value is to be determined, might be .9. We shall see below that the relaxation of the explicit integrality condition on x_i does not depart radically from the desired interpretation of x_i since for some confidence level close to α, an optimal solution to (2) will automatically assign only 0–1 values to the x_i. Since there is nothing sacred about .9, for example, we shall in practice begin with a "judicious" choice of α! To proceed with derivation of an optimal scheduling algorithm from (2), we note that the chance constraint simply states that

$$1 - F_S\left(\sum_{i=1}^{n} s_i x_i\right) \geqq \alpha$$

when the distribution function F_S is given by $F_S(t) = P[S < t]$. Since the x_i are to be chosen as constants and not as functions of S, they are "zero-order" decision rules in the usual terminology of chance constrained programming. It is not difficult to prove, using the monotonicity of F_S and the definition of F_S^{-1} (given by $F_S^{-1}(1-\alpha) = \sup\{t: F_S(t) \leqq 1-\alpha\}$) that the set of values for the x_i which satisfy the chance constraint of (2) is the same as the set of values for which

$$\sum_{i=1}^{n} s_i x_i \leqq F_S^{-1}(1-\alpha) \tag{3}$$

where $F_S^{-1}(1-\alpha)$ denotes the $1-\alpha$ fractile of the distribution function of S. Note that this is true even if F_S is discontinuous or not 1–1. In view of this, (2) is equivalent to the ordinary linear programming problem

$$\text{maximize } \sum_{i=1}^{n} u_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} s_i x_i \leqq F_S^{-1}(1-\alpha) \tag{4}$$

$$\text{and } 0 \leqq x_i \leqq 1.$$

Two observations are pertinent here. First, $F_S^{-1}(1-\alpha)$ provides a "certainty equivalent" for the random space S which will actually be available. The structure of the linear programming problem (4) is so special that an optimal solution is obtainable by inspection, as is well known. Such an optimal solution is determined as follows. Order the variables x_i such that $u_1/s_1 \geqq u_2/s_2 \geqq \cdots \geqq u_n/s_n$ and take $x_1 = 1$, $x_2 = 1$, $\cdots$ until $x_k = 1$ would violate (3). x_k should then be set to the fractional value

$$\frac{F_S^{-1}(1-\alpha) - \sum_{i=l}^{k-1} s_i x_i}{s_k}$$

in order to just use up the remaining space. However, if α happened to be such that

$$F_S^{-1}(1-\alpha) = \sum_{i=1}^{k-1} s_i x_i,$$

we can see that no variable x_i would need to be set to

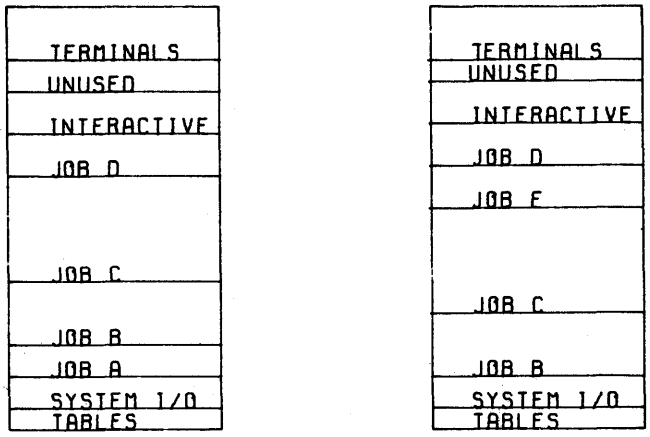


Figure 4—Main memory allocation transition

a fractional value. In view of the approximate nature of the policy restriction in (2), it is not unreasonable to expect this to be the case for some confidence level close to α if not for α itself.

We therefore implement the following scheduling method: schedule jobs in decreasing order of their u_i/s_i values until available space $[F_S^{-1}(1-\alpha)]$ is used up. In practice, each job has associated with it a "job cost" $c_i = s_i t_i$. In this terminology, the scheduling order is determined by the order of increasing costs c_i since $u_i/s_i = 1/t_i s_i = 1/c_i$; this rule is particularly simple in form.

Our formulating the space requirement as a chance constraint has another motivation. If batch jobs were never permitted to delay interactive jobs because of conflicting space demands, corresponding to a choice of $\alpha = 1$, the result would be a high incidence of unused memory space, a smaller number of active batch jobs, and a resulting impairment of CPU efficiency. However, a small batch job (e.g., the last one scheduled) can be swapped to ECS when interactive needs dictate, with a modest reduction in ECS capacity for interactive jobs and in interactive response speed. In practice, when the loaded batch jobs are confronted with a demand for space by an interactive job (corresponding to a violation of the space limitation), the last batch job scheduled is swapped to ECS. In view of a choice of α approximately equal to .9, this will happen only about 10 percent of the time, which is in accord with the policy toward interactive response time and efficient utilization of the CPU.

In summary, by a formulation of the scheduling problem as the maximization of the sum of the completion rates $1/t_i$ of the jobs scheduled, subject to permissible interference with interactive jobs if this interference is sufficiently infrequent, an optimal scheduling policy with extremely simple form can be achieved.

In the next section we consider the effect of this scheduling policy on the memory compacting problem.

MEMORY COMPACTING PROBLEM

In a non-paged memory system with only a single bounds-checking facility per process, the memory allocated to a given job must be continuous. When a job terminates in a multiprogramming non-paged system, the total memory available to the next job scheduled is potentially the sum of all unallocated memory regions. This is the amount assumed in the previous discussion of batch job scheduling, i.e., we assumed that memory compacting was done whenever required. Making this potential total actually available frequently requires that other jobs, which are still running, be moved. We now consider the factors which affect the frequency with which these storage moves are necessary and memory management policies used to minimize the total cost of compacting.

Memory compacting may be necessary whenever a job ceases to occupy its memory. For a batch job, this situation can arise because of a termination or a pre-emption. For an interactive job, the situation is caused primarily by a terminal wait condition or a time slice pre-emption. In the current system, the memory requirements of the batch system change about once every 10 seconds. In the worst case, this will require approximately 30 milliseconds of compacting with one block of ECS dedicated to this application. The effects of the changing memory requirements of the interactive partition on the necessity for memory compacting could be much more serious. If these changes require that batch jobs be moved, the overhead will be substantial. In addition, in the previous section we discussed the possibility of swapping a batch job to and from ECS in order to fill in the valleys of interactive storage requirements and increase the average number of active processes in order to decrease the CPU idle time. If the swapping of this batch job requires that other batch jobs be moved, the overhead could also be large. In order to avoid these problems most of the time and minimize the cost of memory compacting we use the following ordering policy for active processes. All batch jobs are packed into one end of main memory in the order scheduled. Since jobs are scheduled in order of increasing costs, the batch job likely to be swapped because of interactive memory demands normally is last, i.e., closest to the unallocated region. Hence

swapping of this job will not require moving other batch jobs. The interactive control point is placed last. It is next to the unused memory. This unused memory is sufficient to satisfy its demands 90 percent of the time, as described in the previous section. Figure 4 illustrates the insertion policy used by the scheduler to maintain this ordering. Job a, the lowest cost batch job running, terminates. The scheduler decides to load job e whose cost is between the cost of job d and job c. Thus jobs b and c must be moved down to fill the gap left by a and job d must, in general, be moved to either make room for job e or fill an additional gap created by job e. It should be noted that the cost of a job varies with time so that the cost ordering of jobs in memory may change. In practice, this rate of change is very small with respect to the scheduling rate for the interactive control point. When it does happen, the highest cost job will be pre-empted when a pre-emption is necessary, regardless of the ordering in main memory. If this job is then rescheduled, it will be put in the proper place to restore the cost ordering in memory. This policy minimizes moves due to changes in the memory demands of the interactive system but maximizes moves generated by completions in the batch system. However, the average rate of completion of batch jobs is one every 10 seconds. Thus this overhead is still less than .3 percent in practice.

CONCLUSION

A non-paged multi-programming computer system required to support a time-sharing system and a batch-processing system faces the problem of memory compacting and memory demand interference. Proper memory management policies can minimize these difficulties. Measurements of the system and job characteristics provide the basis for an adequate design. We have shown that such a design for a CDC 6600 and CDC Extended Core Storage can support an interactive load of approximately 40 users with a response time of less than a second at very small cost to a highly efficient batch processing system.

ACKNOWLEDGMENTS

This research was sponsored in part by the Control Data Corporation under a research grant to the Computation Center at The University of Texas at Austin and by the National Science Foundation under grant GJ–741.

REFERENCES

1 J FOTHERINGHAM
Dynamic storage allocation in the Atlas computer including an automatic use of a backing store
Communications of ACM October 1961

2 B W ARDEN B A GALLER T C O'BRIEN F H WESTERVELT
Program and addressing structure in a time-sharing environment
Journal of the ACM January 1966

3 P DENNING
Resource allocation in multi-process computer systems
PhD Dissertation Mass Inst of Tech June 1968

4 R E FIKES H C LAUER A L VAREHA
Steps toward a general purpose time-sharing utility using large capacity core storage and TSS/360
Proc of Nat Conf ACM 1968

5 K FUCHEL S HELLER
Considerations on the design of a multiple computer system with extended core storage
Communications of ACM November 1968

6 W ANACKER C P WANG
Performance evaluation of computing systems with memory hierarchies
IEEETEC December 1967

7 D N FREEMAN
A storage-hierarchy system for batch processing
Proc AFIPS SJCC 1968

8 H D SCHWETMAN J C BROWNE
A study of central processor inactivity on the CDC 6600
To be published 1970

9 H N CANTRELL A L ELLISON
Multiprogramming system performance measurement and analysis
Proc AFIPS SJCC 1968

10 U N DE MEIS N WEIZER
Measurement and analysis of a demand paging time sharing system
Proc ACM 1969

11 B W ARDEN D BOETTNER
Measurement and performance of a multiprogramming system
Proc Second Symposium on Operating Systems Principles Princeton New Jersey 1969

12 H D SCHWETMAN J R DELINE
An operational analysis of a remote console system
Proc AFIPS FJCC 1969

13 A L SCHERR
An analysis of time-shared computer systems
Research Monograph No 36 1967 MIT Press Cambridge Massachusetts

14 G E BRYAN
JOSS 20,000 hours at the console—A statistical summary
Proc AFIPS FJCC 1967

15 K FUCHEL G CAMPBELL S HELLER
The use of extended core storage in a multiprogramming operating system
Proc Third International Symposium on Computer and Information Sciences Miami Beach Florida 1969

16 A CHARNES W W COOPER
Deterministic equivalents for optimizing and satisfying under chance constraints
Operations Research Vol 11 1963

17 D P GAVER JR
Probability models for multiprogramming computer systems
Journal of the ACM July 1967

A study of interleaved memory systems

by G. J. BURNETT

Index Systems, Inc.
Boston, Massachussetts

and

E. G. COFFMAN, JR.

Princeton University
Princeton, New Jersey

INTRODUCTION

There is frequently a severe mismatch between achievable processor and memory speeds in today's computer systems. For example, the CDC–7600 has a 27ns (nanosecond) processor cycle time and a 270ns memory cycle time;[1] the IBM–360/91 has a 60ns processor cycle time and a 750 ns memory cycle time.[2] In order to obtain the desired increase in the effective memory speed, an efficient memory system must use such techniques as interleaving memory modules and implementing an automatic level in a memory hierarchy (e.g., a slave memory[3] as in the IBM–360/85[4] or 195[5] and the CDC–7600). In the past, interleaving was often studied by simulation using a random address generating source to obtain memory requests.[6,7] This paper discusses results of mathematical analyses of models of interleaved memory systems. In these investigations the properties of addresses generated by instructions and data have been distinguished.

Interleaving is achieved by dividing the memory into separate, independent modules that can be in simultaneous operation. Information is then stored in the memory with sequential items residing in modules that are consecutive, modulo the number of memory modules; equivalently, the low order address bits specify the memory module number, and the high order bits specify the word within a module.

We will first present a model of interleaved memory systems. In the analysis of this model we obtain a figure of merit for such systems, viz. the average number of memories in operation on data or on instructions during a memory cycle. Results of numerical investigations of this figure of merit, which we call the average memory bandwidth, will be displayed both individually for instruction and data requests and for a combination of these requests into a system structure utilizing interleaving.

ANALYSIS OF A MATHEMATICAL MODEL OF INTERLEAVED MEMORY SYSTEMS

Model and terminology

The model is pictured in Figure 1. There are n identical modules each capable of reading or writing one word per memory cycle. We shall assume that the modules operate synchronously and with identical memory cycle times. In practice the Request Queue contains conventional instruction and data storage addresses; however, for our purposes only the module number from the address is of interest. Thus, we will consider the requests r_i, $i = 1, 2, \cdots$, to be integers from the set $(0, 1, \cdots, n-1)$.

The Scanner operates by admitting new requests to service until it attempts to assign a request to a busy memory module. To do this, prior to the start of a given memory cycle, i.e., during the previous memory cycle, the Scanner inspects the Request Queue beginning with r_1 and determines the maximum length sequence of distinct module requests. That is, it scans the queue to the first repetition of a module request. The memory requests in this maximum length sequence are then sent to the appropriate memory modules so that they will be active in the next memory cycle. The maximum length sequences found in this manner are called *request sequences*, and their lengths can be from 1 to n requests. We shall assume that the Request Queue always contains at least n items when inspected.

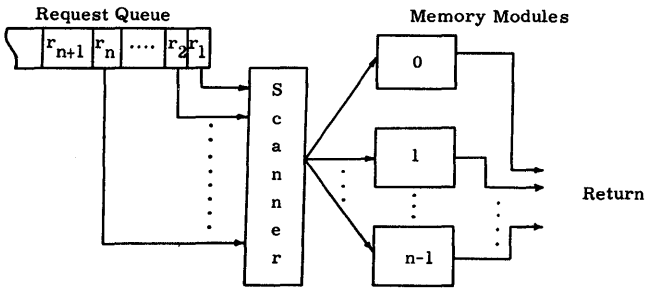


Figure 1—Interleaving model

In effect, the queue will always be saturated and the system operating at capacity. More formally then, a sequence of requests $r_1, r_2, \cdots, r_k$ at the head of the request queue is a request sequence if and only if (1) for all i, j $(1 \leq i \leq k, 1 \leq j \leq k)$ $i \neq j$ implies $r_i \neq r_j$, and (2) there exists a p, $(1 \leq p \leq k)$, such that $r_p = r_{k+1}$.

We will assume that the Scanner can process n requests per memory cycle; this is equivalent to assuming that the processor can handle n words per memory cycle. (We shall shortly consider the case where the Scanner can process $M \leq n$ requests per cycle.)

It is then clear that the effectiveness of an interleaved memory system is determined by the probability density function (pdf) for the lengths of request sequences. Let $P(k)$, $k = 1, 2, 3, \cdots n$, denote this pdf. Then $B_n = \sum_{k=1}^{n} kP(k)$ denotes the mean value of this pdf. B_n will also be called the average memory bandwidth with the units of words/memory cycle. In practice, repetitions in the Request Queue occur sufficiently often such that B_n is considerably less than n, the maximum value for B_n. We will thus be interested in considering systems with M processor cycles per memory cycle where $1 \leq M \leq n$. (Clearly $M > n$ is useless in terms of obtaining information from the memory since we are assuming we have only n memory modules each capable of accessing one word per memory cycle.) Such a system allows a slower processor to be used and yet may offer almost the same average memory bandwidth as a system with $M = n$. The average memory bandwidth for such a system will be denoted by B_M and will also be calculated directly from the $P(k)$ mentioned above.

To compute the $P(k)$ (and therefore B_n), it is necessary to know the properties of the sequences $r_1, r_2, r_3, \cdots$ in the Request Queue. Hellerman[8] has analyzed this model under the assumption that for all i, $\Pr[r_i = j] = 1/n$ for all $j \in S_n$, and that this probability is stationary (i.e., the same for every memory cycle). A simple analysis for this model shows that the probability of a string of k distinct integers followed by a repetition of one of these k is given by

$$P(k) = \frac{(n-1)(n-2)\cdots(n-k+1)}{n^{k-1}} \frac{k}{n} \quad (1)$$

$$= \frac{k(n-1)_{k-1}}{n^k}; \qquad 1 \leq k \leq n$$

where we use the notation[9]

$$(i)_j = i(i-1)(i-2)\cdots(i-j+1); \; (1 \leq j \leq i)$$

and

$$(i)_0 = 1$$

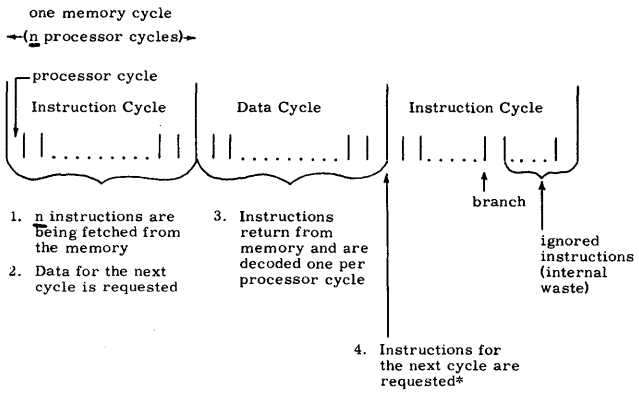
Hence,

$$B_n = \sum_{k=1}^{n} kP(k) = \sum_{k=1}^{n} \frac{k^2(n-1)_{k-1}}{n^k} \quad (2)$$

Hellerman has carried out curve fitting to get the approximation $B_n \simeq n^{.56}$, $1 \leq n \leq 45$, which is accurate to within about 4 percent.

In his model, Hellerman assumed that instruction and data requests were intermixed in the Request Queue. Inasmuch as successive instruction requests tend to have more serial correlation than successive data requests, we have chosen to represent instruction and data request sequences separately in our model. In addition we will investigate a system structure in which these requests are handled separately. Thus, we consider the model of Figure 1 to contain two separate queues, the Instruction Request Queue and the Data Request Queue. In this paper, we will only consider a system structure that alternates instruction and data cycles. That is, for one memory cycle the Scanner obtains all requests from the Instruction Queue and for the next cycle only the Data Queue is scanned. We refer to this system structure as the Instruction Data Cycle Structure, IDCS.

The above operation enables us to carry out separate mathematical analyses in order to determine the average memory bandwidth for an instruction cycle, IB_n, and for a data cycle, DB_n. B_n is obtained from a weighted sum of IB_n and DB_n. The weighting depends on the assumed percentage of the instructions that request data and as a result on the actual values of IB_n and DB_n. Studies of program composition[10] suggest that approximately 80 percent of all instructions require an operand (data) reference to storage. We will make the somewhat more conservative assumption that 80 percent of the instructions executed, excluding branch instructions, request data. We shall use this as an assumption to calculate B_n from DB_n and IB_n.



* Note that all n sequential instructions can be requested at once since they are for sequential words that are located in sequential memory modules.

Figure 2—Instruction processing

Instruction model

The efficient use of an interleaved memory in a single processor system is predicated on the existence of a processor fast enough to handle several instructions per memory cycle. In order to obtain this multiple instruction capability, each memory cycle the processor requests a number of instructions beyond the present instruction counter address, i.e., it looks ahead. Therefore, such a system must have buffering for instructions, for data (operands), and for storage addresses.

For the purpose of determining the memory bandwidth we only need to assume that the instruction buffer holds the maximum number of instructions that can be obtained from the memory during a memory cycle, n, and that the instruction decoding unit can decode all these instructions in one memory cycle. (In other words, the system is capable of decoding one instruction per processor cycle.) We also assume that a branch instruction requested during one memory cycle will be decoded in the next memory cycle, and will immediately affect the instruction stream.

Under the foregoing assumptions, our system operates as shown in Figure 2 and as described below.

1. At the beginning of an instruction cycle the Scanner requests the n sequential instructions following the present instruction counter address. Thus, n memory modules are busy during this cycle.

2. If an instruction requesting a branch is decoded during the next data cycle, the n sequential requests for the next instruction cycle will start from the branch address, i.e., the instruction counter will be loaded with the branch address. If a branch is not found, the next instruction cycle will request n sequential words starting with the word following the last requested instruction.

With the Instruction Queue we associate the parameter λ, the stationary and independent probability that any given instruction generates a branch. Observe that in the data cycle following an instruction cycle in which a branch was requested, all instructions that were requested after the branch will not be decoded. (This is all right since the look-ahead mechanism only guessed that n sequential instructions would be used.) The instructions that are requested but not decoded in a given memory cycle are referred to as *internal waste*. With λ given, internal waste is accounted for in $P(k)$, the probability that k of the n instruction words obtained in an instruction cycle are actually decoded.

$$\begin{aligned} P(1) &= \lambda \\ P(k) &= (1-\lambda)^{k-1}\lambda; \qquad 1 < k < n \\ P(n) &= (1-\lambda)^{n-1} \end{aligned} \tag{3}$$

Note that $P(n)$ is obtained by observing that if the first $n-1$ instructions are not branches then we obtain the maximum number of modules, n, regardless of the actual length of this string of sequential instructions.

IB_n is obtained from:

$$\begin{aligned} IB_n &= \sum_{k=1}^{n} kP(k) \\ &= \lambda + 2\lambda(1-\lambda) + 3\lambda(1-\lambda)^2 + \cdots \\ &\qquad + n(1-\lambda)^{(n-1)} \end{aligned} \tag{4}$$

which can be reduced to:

$$IB_n = \frac{1-(1-\lambda)^n}{\lambda} \tag{5}$$

This is the desired formula for the average memory bandwidth in the IDCS model.

Data model

We use the following model for data request sequences. (This is a generalization of Hellerman's model applied only to data requests.) The first data request in the Data Queue addresses a module at random; thereafter, the ith request ($i \geq 2$) addresses the next module in sequence (modulo n) with stationary probability α, and addresses any one of the modules out of sequence with equal probability

$$\beta = (1-\alpha)/(n-1).$$

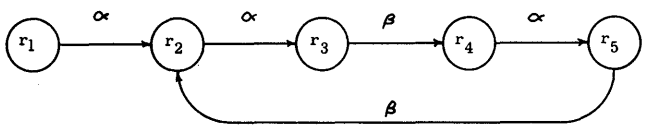


Figure 3—Request graph

Thus, for example, the sequences of requests 0, 2, 3, 2 and 1, 2, 3, 1 would have probabilities $(1/n)\alpha\beta^2$ and $(1/n)\alpha^2\beta$, respectively, in a system having at least $n = 4$ modules. Letting $r_1, r_2, \cdots r_n$ denote the contents of the Data Queue prior to the start of a memory cycle we assume:

$$\Pr[r_1 = k] = \frac{1}{n} \qquad 1 \leq k \leq n \quad (6)$$

$$\Pr[r_{i+1} = (r_i + 1) \bmod n] = \alpha \qquad 1 \leq i < n \quad (7)$$

$$\Pr[r_{i+1} \neq (r_{i+1}) \bmod n] = \beta \qquad 1 \leq i < n \quad (8)$$

With this model we address the problem of computing for the Data Request Queue the probability of request sequences of length k, $Q(k)$. (We have substituted $Q(k)$ for $P(k)$ in order to avoid confusion between instruction and data results.) A recursive enumeration procedure has been developed using request graphs to represent sequences of data requests. An example of such a graph is shown in Figure 3. This particular graph is representative of the case where $k = 5$, and where the sixth request is a repeat of the second request r_2. One set of module requests corresponding to this graph is $r_1 = 4$, $r_2 = 5$, $r_3 = 6$, $r_4 = 2$, $r_5 = 3$. (Note that an α probability connecting two nodes requires that they are sequential requests.) By our definitions of the Data Queue model this set of requests has probability $(1/n)\alpha^3\beta^2$. In fact, all sets of requests corresponding to a given request graph have the same probability. For a given k, there are a number of possible request graphs corresponding to varying the probabilities between the requests and varying the request that is repeated at position $k + 1$ in a sequence. Thus $Q(k)$ can be determined by counting the number of request sequences corresponding to each possible request graph of length k, multiplying the counts by the probability associated with each graph, and then summing these values for all graphs. The enumeration procedure developed carries out the above operations by using counts for graphs of length $k - 1$ to determine the counts for graphs of length k. A complete discussion of the enumeration procedures can be found in reference 11; in the material that follows we will use the results of calculations of $Q(k)$ in order to display numerical investigations involving $DB_n = \sum_{k=1}^{n} kQ(k)$.

Maximum memory bandwidth values

The Instruction and Data Queue models developed above used a formula for IB_n and DB_n which assumes that the maximum memory bandwidth, M, is equal to n. However, in many systems the memory cycle accommodates fewer than n processor cycles. For example, the CDC–7600 has 10 processor cycles per memory cycle, or a maximum memory bandwidth of 10; whereas n can be as high as 32. In a typical system M would be determined by economic constraints. That is, building a memory or processor faster than certain speeds may require much greater complexity or a new, more sensitive or expensive technology. Therefore, it is necessary to know what would be gained by an increase in memory and processor speed or an increase in the speed of one with respect to the other. To gain insight into this latter trade-off in an interleaved system we will consider $M \leq n$.

For $M = n$ we assumed that the Scanner scanned the memory request queue until it found the first repetition. With $M < n$, we will assume that the Scanner scans the queue until either M distinct requests have been found or the first repetition is found. Thus since at most M memory modules will be active in any memory cycle, we have the following extensions to our earlier definitions of IB_n and DB_n:

$$IB_M = \sum_{k=1}^{M} kP(k) + \sum_{k=M+1}^{n} MP(k) \quad (8)$$

$$= \frac{1 - (1 - \lambda)^M}{\lambda}$$

$$DB_M = \sum_{k=1}^{M} kQ(k) + \sum_{k=M+1}^{n} MQ(k) \quad (9)$$

where the procedures discussed earlier are used to calculate $P(k)$ and $Q(k)$. Note also that both IB_M and DB_M are always less than or equal to M, the maximum memory bandwidth.

NUMERICAL STUDY OF INTERLEAVING

To study the performance gain that can be expected from interleaving, we shall examine the following

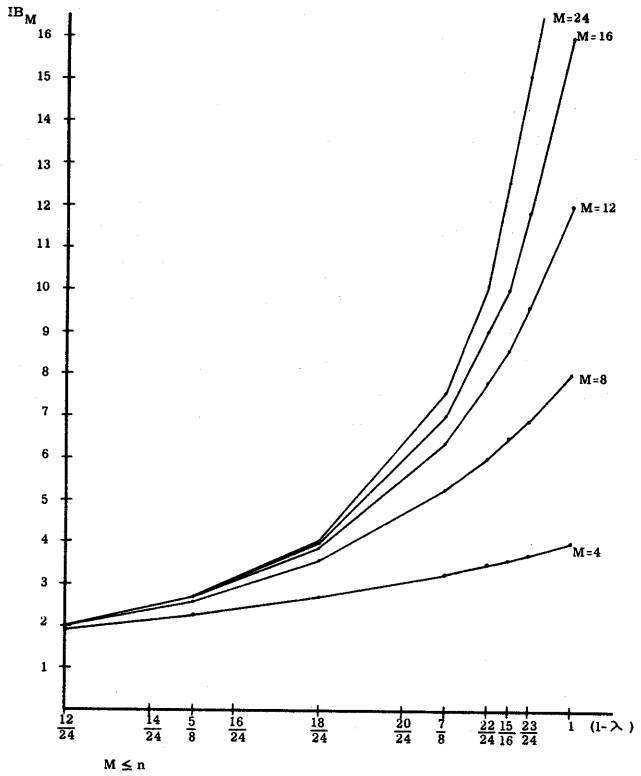


Figure 4

curves:

IB_M vs. $1 - \lambda$

IB_M vs. M

DB_n vs. α

DB_M vs. M

These four curves are sufficient to see the effects of the parameters on instruction and data bandwidths. They are shown in Figures 4, 5, 6 and 7.

Figure 4 is a plot of IB_M versus $(1 - \lambda)$. For higher values of $(1 - \lambda)$—less branching—IB_M rises sharply to its maximum, M. In particular, programs with moderate to high values of $(1 - \lambda)$, i.e. $(1 - \lambda) \geq \frac{7}{8}$, make good use of the interleaved memory system. A detailed analysis of various program mixes would be necessary to determine the most likely range of λ for any particular mix. However, some preliminary analysis shows that values of λ between $\frac{1}{8}$ and $\frac{1}{16}$ are likely for a range of programs, and thus these programs make good use of the memory in a system structure that groups instruction requests, e.g., the IDCS.

When designing a computer system economic constraints require that a good choice of relative memory and processor speeds be made. If the memory cycle time, mc, is fixed by technology considerations then increasing the ratio M = (memory cycle time)/(processor cycle time) corresponds to decreasing the processor cycle time, pc. Figure 5 shows a plot of IB_M versus M with mc fixed. For the lower values of $(1 - \lambda)$, due to internal waste there is a distinct flattening of the curves as M is increased. Thus, for a given value of λ, it is of little value to decrease the processor cycle time beyond a certain point.

Figure 6 shows a plot, for various values of n, of the mean memory bandwidth, DB_n, versus the parameter α. (These and subsequent results for the Data Queue are obtained from reference 11.) Substantial improvements in DB_n are obtained only in the region, $.5 < \alpha < 1.0$. The minimum points in the graph correspond to $\alpha = 1/n$, i.e., random addressing.

Precise specification of α for any given program mix would require program analysis not yet undertaken. From preliminary analysis, however, α between .125 and .5 appears typical.

Figure 7 shows DB_M vs. M for several values of α. The memory cycle is assumed fixed. From the point of view of practical design, it is clear from our earlier remarks that the lower values of M correspond to less

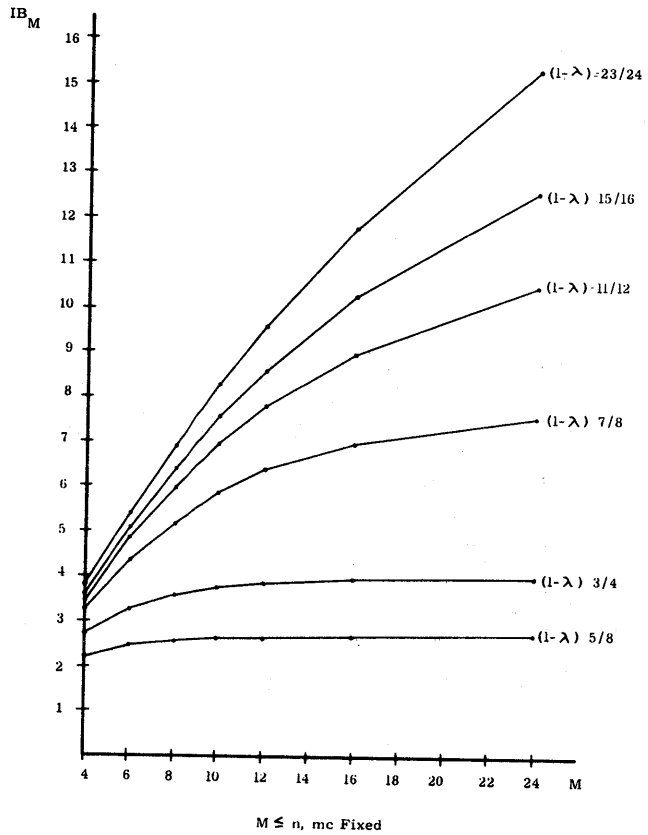


Figure 5

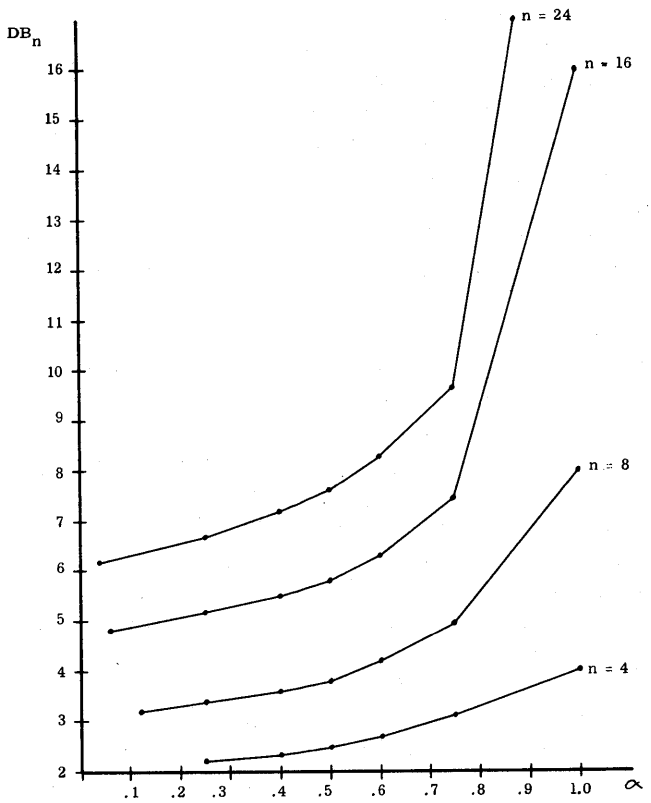


Figure 6

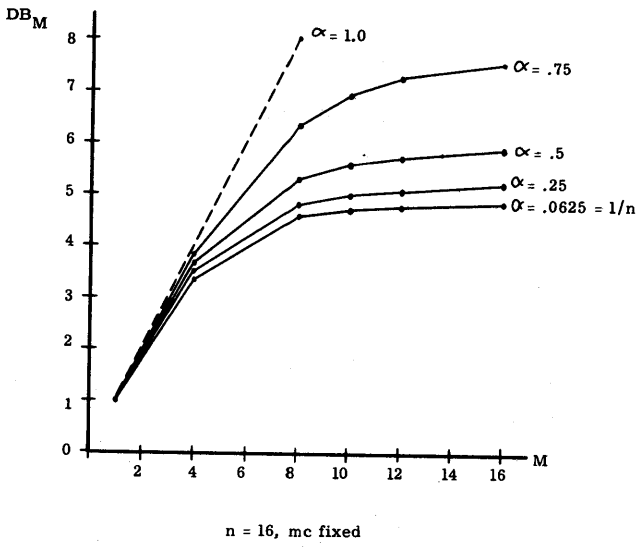


Figure 7

expensive systems since the processor speeds need not be as great relative to memory speeds. It is then interesting to observe that for $\alpha < .5$, M can be as small as $n/2$ without reducing the mean bandwidth more than 15 percent below its maximum value. This effect is the result of increased interference among data requests for larger values of M; thus, for M much larger than $n/2$, the probability of a blockage, i.e., a request asking for a busy module, is very high. Consequently, increasing M beyond a certain point is of marginal value in improving memory bandwidth.

Figure 6 gives some indication of the usefulness of increasing n for a given α; this effect is made clearer in Figure 8, where we have plotted DB_M versus n, for

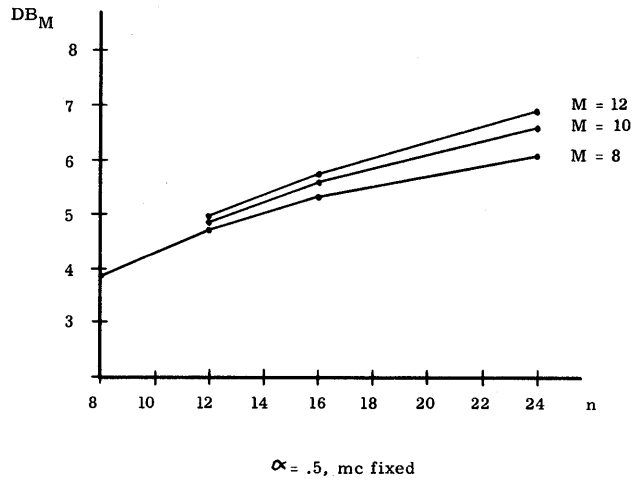


Figure 8

M as a parameter and for α and mc fixed. The behavior for the other values of α was similar.

To summarize, we have observed the following significant features with respect to instruction and data requests in an interleaved memory system: (1) the value of interleaving for programs that have a high α or a small λ; (2) the slow increase of DB_M (and to a lesser extent IB_M) with decreasing processor cycle time once M is sufficiently large; and (3) the increase in DB_M as n is increased. Next we will use these observations in the investigation of a system structure that takes advantage of the interleaved memory system as modeled.

Instruction data cycle structure

We recall that IB_M and DB_M as calculated are for a system structure, the IDCS, in which we have assumed

that alternate memory cycles are allocated to instruction accessing and data accessing. This operation was described earlier and depicted in Figure 2. We will derive a method of calculating B_M for this structure that takes into account the actual dependence of the Data Queue on the Instruction Queue. As stated earlier, this dependence is specified by our assumption that .8 of the executed instructions add requests to the Data Queue. We now make the important observation that the advantage of this structure is that it places into blocks requests that are likely to be sequential. Thus the system utilizes interleaving to obtain the increased average memory bandwidth observed for instructions and data in the previous section. In addition, within a memory cycle there is no interference between data and instruction requests; thus this structure can achieve a higher average memory bandwidth than, for example, structures that use the same Scanner but alternate instruction and data requests within a memory cycle.

Because instruction and data requests are interrelated, we note that, depending on the values of λ and α, either instruction accessing or data accessing can limit the average memory bandwidth. That is, given M and α, there is a sufficiently large λ beyond which fewer than DB_M items will be added to the Data Queue per data cycle, i.e., the Data Queue will not be saturated. In this case of limited instruction accessing, the average data bandwidth is simply the average number of items added to the Data Queue per memory cycle. B_M can then be calculated by taking half the sum of IB_M and this latter average. On the other hand, data accessing will limit B_M if for M and α given, λ is small enough

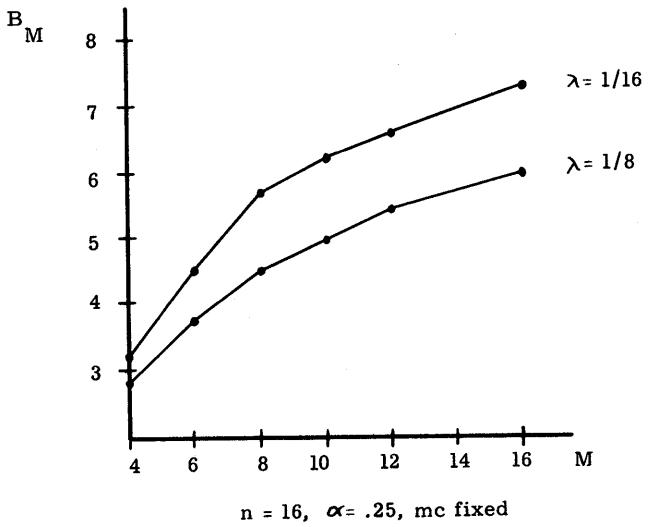


Figure 9—IDCS

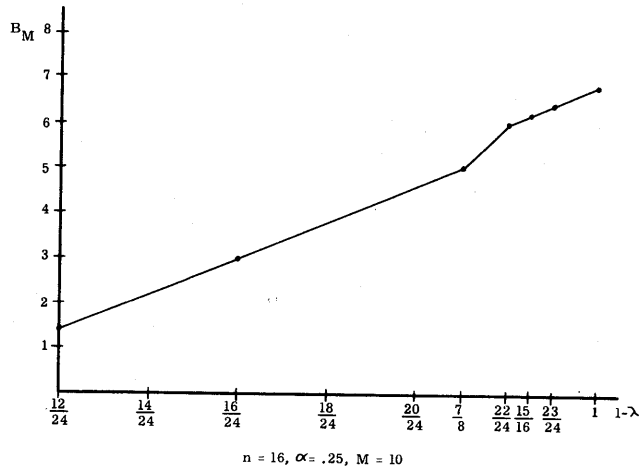


Figure 10—IDCS

such that on the average more than DB_M requests are added to the Data Queue per data cycle. In this case the system must periodically add an extra data cycle in order to avoid overflowing any amount of buffering allocated to the Data Queue. A formula for calculating B_M in this case can be derived from these statements; but these calculations are not shown here since our present interest is simply in the curves resulting from these calculations.

Although the IDCS can be shown to be efficient for a wide range of n values, we shall study $n = 16$ here. Moreover, in Figures 9 and 10 we choose λ and α values in the most likely ranges discussed earlier, namely $\alpha = .25$ and $\lambda = 1/8$ and $1/16$. Comparing Figure 9 to Figure 5, we note that the slope after each change in M is less here than for the corresponding curve from Figure 5. This is to be expected since only .8 of the executed instructions request data and since the Data Queue saturates for $M \geq 8$ (see Figure 7). Figure 10 shows B_M vs. $(1 - \lambda)$ for $M = 10$. Here again we can observe the limiting effect of data accessing on B_M as λ is increased. In particular, for $(1 - \lambda) \sim 11/12$ the Data Queue saturates and thus restricts further substantial increases in B_M with increases in λ.

SUMMARY

We have discussed an analysis of interleaved memory systems where the distinct properties of instruction and data requests have been considered. The figure of

merit used to investigate interleaved systems was the average memory bandwidth. Numerical investigations of models for the Instruction Request Queue, the Data Request Queue, and the IDCS demonstrated the trade-off between instruction and memory cycle speeds and also showed the significant value of separately grouping instruction and data requests when accessing the memory. This latter grouping of requests substantially increases the average memory bandwidth. Although we have not investigated it here, we note that additional increases in bandwidth can be achieved by reading multiple words from each memory module instead of one word as discussed above. This technique is particularly useful for increasing IB_M if λ is small.

REFERENCES

1 Control Data 7600 Computer System Preliminary Reference Manual Control Data Corporation 1968

2 M J FLYNN P R LOW
The IBM system/360 model 91: Some remarks on system development
IBM Journal of Research and Development Vol 11 No 1 January 1967

3 M V WILKES
Slave memories and dynamic storage allocation
IEEE Transactions on Electronic Computers Vol EC-14 No 2 April 1965

4 C J CONTI ET AL
Structural aspects of the system/360 model 85
IBM Systems Journal Vol 7 No 1 1968

5 R A McLAUGHLIN
The IBM 360/195
Datamation October 1969

6 W BUCHHOLZ
Planning a computer system project stretch
McGraw-Hill New York 1962

7 L J BOLAND ET AL
The IBM system/360 model 91: Storage system
IBM Journal of Research and Development Vol 11 No 1 January 1967

8 H HELLERMAN
Digital computer system principles
McGraw-Hill New York 1967

9 W FELLER
An introduction to probability theory and its applications Vol 1
John Wiley and Sons New York 1968

10 D W ANDERSON ET AL
The IBM system/360 model 91: Machine philosphy and instruction handling
IBM Journal of Research and Development Vol 11 No 1 January 1967

11 G J BURNETT
Performance analysis of interleaved memory systems
PhD Thesis Princeton University 1969

A computer system for bedside medical research*

by STEVEN E. WIXSON, EUGENE M. STRAND and H. WILLIAM PERLIS

The University of Alabama Medical Center
Birmingham, Alabama

INTRODUCTION

The University of Alabama Myocardial Infarction Research Unit (MIRU) supports clinical research on patients who have sustained a myocardial infarction (heart attack). MIRU is a contract from the National Heart Institute whose goal is the reduction of mortality and morbidity from myocardial infarction. Patient rooms provide the environment for intensive coronary care and research. Two laboratories facilitate study of critically ill patients with complicating conditions, such as, shock, congestive heart failure and severe arrhythmia. The digital computer housed adjacent to the patient rooms is dedicated to on-line real-time MIRU research (see Figure I., MIRU).

The Shock Research Unit of Los Angeles County Hospital pioneered the application of digital computers for on-line clinical research of cardiovascular functions.[1] Following that effort, electronic data processing techniques are being used increasingly for patient monitoring.[2–5] These applications emphasize clinical care of postoperative patients, and they primarily monitor only a few variables, such as, blood pressure, heart rate, respiratory rate, temperature, and urine flow. The patient monitoring programs developed by Sheppard, et al., at the University of Alabama[5] are used in the clinical care of patients in the MIRU.

The variety of MIRU research protocols (e.g., thermal dilution cardiac output, assisted circulation, ECG rhythm analysis) demand a changeable support system both in the bedside instrumentation and in the computer software. The MIRU research system emphasizes flexibility in facility allocation and ease of programming and operation.

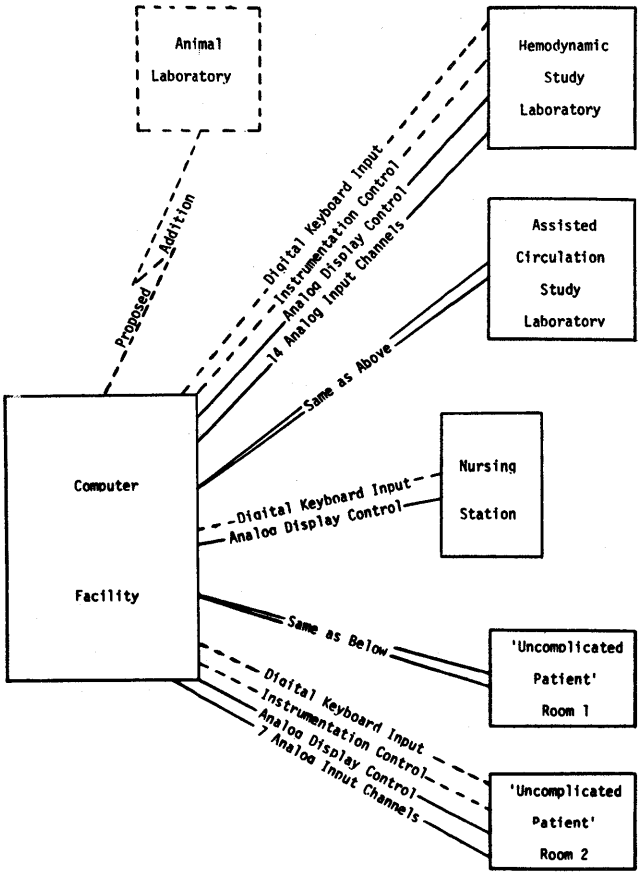


Figure 1—Myocardial infarction research unit

Computer Requirements

MIRU research requires the following computer capabilities:

- Acquisition and analysis of data from multiple beds,

* Supported in part by U.S. Public Health Service Contract No. PH43-67-1441

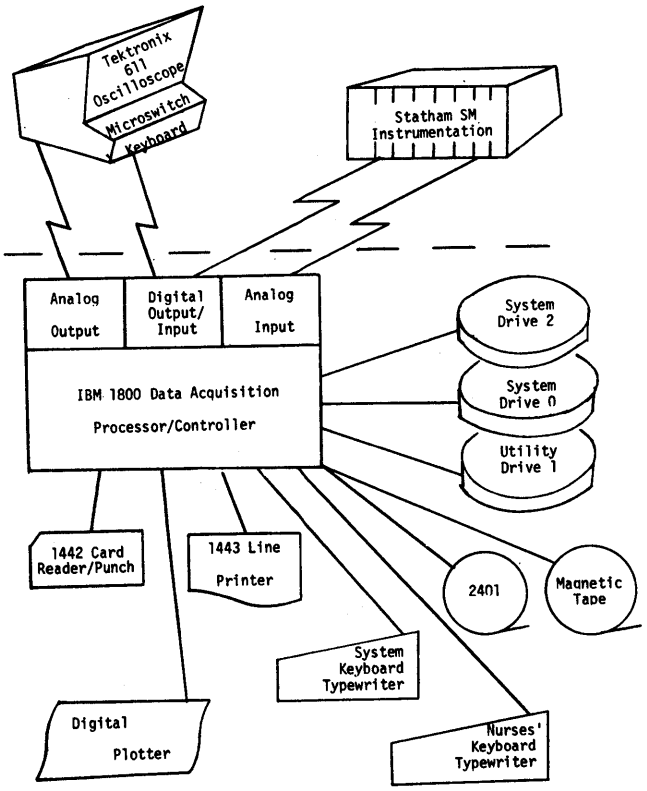


Figure 2—System design

- Control of measurement and therapy devices,
- Real-time display of information, and
- Concurrent development of computer programs.

The patient data for research studies include physiological, historical, physical examination, laboratory, clinical observation, intervention, and pathological data. Intervention data include medication, therapy, changes in position, and research protocols. Physiological data acquired in analog form include sampled data and derived parameters.

Computer configuration

The IBM 1800 Data Acquisition and Control System[6] with the Multiprogramming Executive (MPX)[7] provides the foundation for the system which meets these requirements. Supplementing MPX, a MIRU executive supports data entry at multiple terminals and allocates system facilities to multiple patients. This executive requires minimal alteration to MPX.

The IBM 1800 processor-controller is a 32K word, 2-microsecond cycle time computer. Each word is 16 bits plus 2 bits for parity checking and storage protection. Digital input, digital output, and analog output features allow connection of special devices, e.g., remote terminals (see Figure 2., System Design).

Three direct access disk drives provide 1.5 million words total of on-line storage for programs and data (see Table I, Disk Allocation). Two drives are reserved for the real-time parts of the MIRU system. The third is a utility drive serving as one of the following:

- backup for the system drives,
- storage for source and object programs during development,
- storage of the MIRU master patient index for statistical studies, and
- large volume disk storage for a particular research protocol.

Two 60kb tape drives provide high volume storage for research studies. One, the real-time tape drive, logs data from patient files for retrospective study. The other, the special study tape drive, provides temporary data storage in scheduled research procedures and backup for the real-time tape drive.

Two 20kc analog-to-digital converters provide continuous and noncontinuous modes for analog data acquisition. One, the real-time converter, converts data

TABLE I—Disk Storage Allocation

SECTORS	*DRIVE 0*	*DESCRIPTION*
300		IBM system programs.
800		TASK working storage.
293		Coreload storage.
60		Temporary patient file.
2		Terminal control file.
54		Interrupt save area.
27		Batch save area.
56		Executive Director.
8		Cold start program.
SECTORS	*DRIVE 1*	*DESCRIPTION*
24		Disk index table.
350		IBM relocatable programs.
100		General relocatable subroutines.
286		Batch work storage.
40		Test process work storage.
50		Test coreload area.
750		Source program file.
SECTORS	*DRIVE 2*	*DESCRIPTION*
8		Disk index table
1000		Patient active file.
592		Coreload area.

which is processed continuously by core-resident routines. The second, the special study converter, is used for converting bursts of data for routine clinical processing and research experiments. MPX schedules the second converter by queueing conversion requests. The queued requests are serviced according to a priority assigned to the research experiment. Experiments demanding immediate response receive a high priority. The special study converter serves as back-up for the real-time converter.

Multiprogramming executive

The IBM 1800 Multiprogramming Executive operating system is the real-time monitor for the computer. MPX provides automatic handling of interrupts from data input–output (I/O) devices and user sources, automatic program scheduling, on-line hardware diagnostics, and time sharing for real-time routines, process programs, and background processing. The MIRU system provides these areas for program execution: Special coreload area (SPAR), coreload area, and variable core. These areas service, respectively, programs of high response and short execution (1 millisecond), medium response and medium execution (1 second), and slow response with variable execution (see Table II, Core Allocation).

The MIRU executive features: (1) Remote terminal control of system facilities, (2) Computer controlled medical instrumentation, (3) Task concept for allocation of system facilities, (4) FORTRAN programming environment with multiple entry points, (5) Flexible program communication including program control of exception conditions, and (6) Standardized handling for data storage in the patient file.

TABLE II—Core Storage Allocation

PARTITION OR AREA	*SIZE*	*DESCRIPTION*
Inskel Common	3 K*	Core resident storage area for program communications. Terminal control block, bed and parameter control blocks, task block and working storage.
Executive I/O and Director	10 K*	Core resident parts of the IBM MPX System. Interrupt Handler, Program Scheduler, Disk I/O Routine, Error Routine.
MIRU Executive and FORTRAN Subroutines	3 K*	Keyboard entry routine, instrumentation handler, MIRU housekeeping routine, task timer control. Frequently used FORTRAN subroutines.
SPAR	4 K*	Special coreload area for fast response (Millisec). Continuous signal processing.
Core Load Area 1	4.5 K	Medium response (1–5 sec). Short execution times (Up to 1 sec). Disk loaded programs to handle remote terminals.
Core Load Area 2**	4 K	Slow response (10–20 sec). Longer executing time (5 sec). Work horse area used by most MIRU processing programs.
Variable Core	7.5 K	1st priority—high response processing on a core exchange basis for programs too large for Areas 1, 2. 2nd priority—long executing, large programs with no response required. 3rd priority—batch processing.

*These areas are storage protected.
** This area is planned for the future when additional core storage can be obtained. These programs are now executed as 1st priority in variable core.

REMOTE TERMINALS

Remote terminals control the work load of the disk oriented computer system. The remote terminal consists of a storage oscilloscope and a keyboard for display and entry of information. The storage oscilloscope produces excellent graphic and alphanumeric displays for review of information. High quality graphic plots are essential in the MIRU environment where large quantities of analog data are processed. The storage oscilloscope also offers visual quality control of signals sampled by the computer.

The keyboard keys and lights are connected to the computer's digital input and output points. Data entries (numerics, minus sign, blank, decimal point) are displayed on the top line of the scope for visual verification as the key is depressed. Action keys (clear, enter, respond, reset) cause the computer to perform the specified function. The lights (attached, busy, message) show the terminal's status.

The keyboard and oscilloscope terminals facilitate communication between programs and researchers. Through audio-tone and lights on the terminal, alarm conditions, alert conditions, and routine messages can

TABLE III—Terminal Control Block (TCB)

WORDS	*P**	*DESCRIPTION*
1	P	Terminal number.
2	P	Address of task block in control of terminal.
3	P	Bed control block number.
4	P	Hardware bit mask for lights and display.
5–8		Display "mode" scale factors.
9–10		Display origin for keyboard entry.
11–13		Display origin for FORTRAN IOCR.
14–21		Display scale factors.
22–31	P	Special function program names.
32–47		Keyboard buffer and pointer.
48		Time of data entry.
49	P	Digital input address.
50		Reserved.

* NOTE: "P" means storage protected, and a blank space means not protected.

be signalled. The user can enter data at the keyboard for a program, and the program can display textual, numeric, and graphic information.

A fixed in-core table of parameters exists for each terminal. This terminal control block (TCB) contains information pertinent to the generating of displays and keying of data. MIRU executive plot routines reference the TCB for scaling factors. The keyboard entry routine buffers characters in the TCB (see Table III, Terminal Control Block).

The patient's bedside or nurses' station terminal can be used to call programs into execution and to enter data during program execution. A terminal is normally in program call mode (unattached). A program request through a MIRU executive subroutine dedicated (attaches) a terminal for data entry.

A three-digit name identifies each process program. The first digit is the hardware priority[6] level at which the program will execute. Digits 2 and 3 provide identification for programs which execute on that level. Entry through the keyboard of a program name queues the program for execution. Function buttons on the keyboard map through the TCB into the ten programs most frequently called from a particular terminal. Selecting a function button queues a specific program for execution. Thus a program can be selected either by keying the three digit name or by selecting a function button.

When a keyboard has been attached to a program, data can be entered for that program. Up to 15 characters of information (numerics, minus sign, blank, decimal point) can be buffered in the TCB. Individual data words are separated by the blank, so more than a single value can be entered. For example, the string [5 12.7 1 −53 15] represents five distinct data entries. MIRU executive subroutines move the character string from the TCB and convert it to FORTRAN real or integer values.

BEDSIDE INSTRUMENTATION

Commercially available medical instrumentation provides 7 to 14 channels of analog signals from each patient room. The instrumentation has been modified to permit computer identification of transducers and modules and to allow computer control of bedside devices. The modular computer-controlled instrumentation meets the research requirements of MIRU, as the bedside instrumentation requirements vary with different protocols being conducted. In patient rooms, computer-connected cabinets accept up to 7 channels of instrumentation. In the two laboratories, up to 14 computer-connected module positions are available (see Figure 1, MIRU). Transducer panels above the patient's bed provide one computer-coded transducer connector for each module position in the cabinet. Above each connector is the "transducer active" button used to signal the computer to put a transducer on-line.

When a analog signal is needed, the nurse connects the transducer, plugs in an amplifier module, and presses the transducer active button. The computer reads digital information from the amplifier module and transducer connector. A computer program checks the 5-bit code to insure proper setup of the instrumentation. Using the remote terminal, the program guides the calibration of the amplifier. A light in the "transducer active" button signifies the signal is on-line.

A second pressing of the button sets the signal off-line removes the calibration tables, and turns off the transducer active light.

In the computer, tables store module addressing and calibration information for all programs using a signal. A bed control block (BCB) for each patient in the research unit contains identification and physiological information used frequently by programs. Programs access the data in the BCB through MIRU executive routines which load or save values. A program can only address the BCB of the bed for which it is active.

Since instrumentation requirements vary, a block of storage called the parameter control block (PCB) is dynamically allocated when a parameter (e.g., blood pressure, surface ECG) is placed on-line. A fixed section of the PCB contains addressing and calibration information. A variable section holds the derived data related to the parameter. The derived values are usable

for all programs and are referenced through system load/store routines (see Table IV, BCB's and PCB's).

TASK CONCEPT

The basic unit of work in the MIRU system is the 'task.' A task is defined as a disk loadable program that is active for a specific patient and is uniquely identified by a program name and a bed number. It may be of short duration (a few hundred milliseconds), such as, a summary display; or it may be of long duration (the entire patient stay), such as, a monitoring program executed at periodic intervals. Tasks can be initiated from the remote terminals or through other tasks.

Since a new task is started for each patient, one program is serially reusable for all patients on the unit. This conserves disk storage; since only one copy of a program need be stored on disk.

The task concept dynamically provides system facilities to each patient in the research unit. The bedside terminal will be dedicated to any task for entry or display of information. Core and disk working storage are dynamically allocated for intermediate storage of parameters. Lights and audio alarms are available for signalling alarm and message conditions. Real-time measurements from parameters are available to all programs, and any program can log information to the patient file.

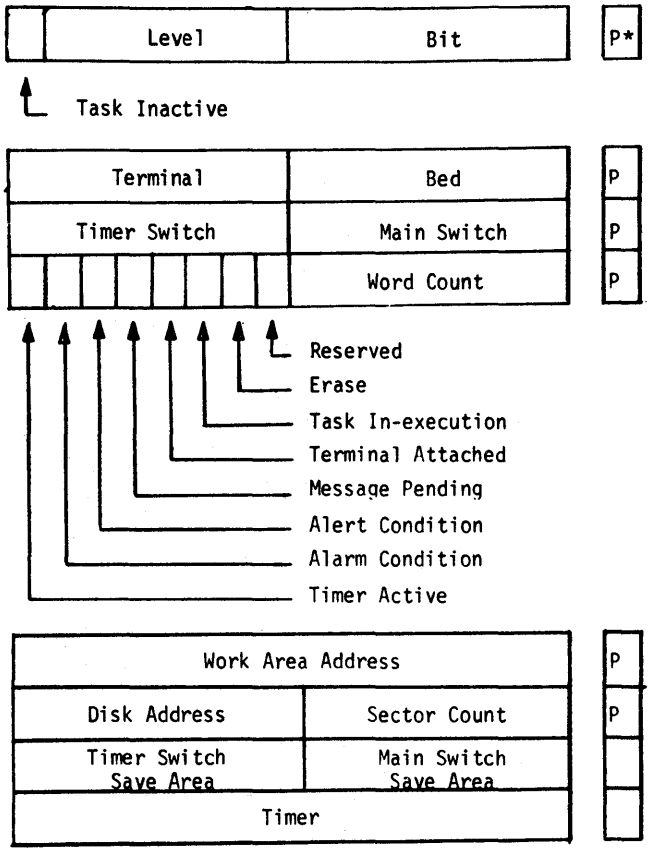
A programmable timer is provided for each task. The 1-sec. time base timer can be used to recall a program at a set interval, check data entry or response to alarm conditions, or schedule a program execution on a periodic basis.

At task request time, the MIRU executive allocates eight words of core called a task block and initializes task parameters. The parameters include the program name, the terminal number, the bed number, two program switches, eight program status flags, the word count and address of in-core storage, the sector address and count of disk storage, a program switch save area, and a task timer (see Figure 3, Task Block).

TABLE IV—Bed Control Block (BCB) and Parameter Control Block (PCB)

WORDS	*P**	*BED CONTROL BLOCK DESCRIPTION*
0	P	BCB length.
1	P	Bed control block number.
2–5	P	Patient number.
6–12	P	Patient name.
13	P	Digital input/output addresses.
14–40		Physiological data storage.
41	P	Maximum number of PCB's.
42–**		Parameter block addresses.
WORDS	*P**	*PARAMETER CONTROL BLOCK DESCRIPTION*
0	P	PCB length.
1	P	Multiplexor address of module.
2	P	Address of task block in control of module.
3	P	Slope calibration.
4	P	Zero calibration.
5+		Derived physiological parameters.

* NOTE: "P" means storage protected, and a blank space means not protected.
** NOTE: = 73 for laboratories.
= 47 for uncomplicated patient rooms.



*NOTE: "P" means storage protected, and blank space means not protected.

Figure 3—Task block

Figure 4—MIRU Sample Task Program

```
SAMPLE TASK PROGRAM
// FOR STASK
C*************************************************************
C*    NAME          STASK                                    *
C*    TITLE         SAMPLE MIRU TASK PROGRAM                 *
C*    DATE          4/1/70                                   *
C*    FUNCTION      THIS ILLUSTRATES HOW PRO-                *
C*                    GRAMS ARE STRUCTURED IN                *
C*                    THE UNIVERSITY OF ALABAMA              *
C*                    MIRU SYSTEM. THE WRITEUP               *
C*                    IN THE MIRU PROGRAMMERS                *
C*                    GUIDE EXPLAINS THIS PRO-               *
C*                    GRAMMING IN DETAIL.                    *
C*    ENTRY         CALL TASK (NAME, IERR) FROM              *
C*                    ANOTHER PROGRAM OR CALL                *
C*                    FROM REMOTE TERMINAL                   *
C*                    (PROGRAM SWITCH SET TO 1)              *
C*                    CALL TASKP (NAME, WDCNT,               *
C*                    NOPAR, ARRAY, IERR) FROM               *
C*                    ANOTHER PROGRAM (PRO-                  *
C*                    GRAM SWITCH SET TO 2) CALL             *
C*                    TENTR (NAME, ICODE, IERR)              *
C*                    BY ANOTHER PROGRAM (PRO-               *
C*                    GRAM SWITCH SET TO 3)                  *
C*                  ERROR RECALL (PROGRAM                    *
C*                    SWITCH SET TO 4)                       *
C*    EXIT          CALL MEXIT (TYPE, NEXT                   *
C*                    SWITCH, TIMER SWITCH,                  *
C*                    TIMER INTERVAL)                        *
C*************************************************************
      DATA NAME /Z***/
C*****PROGRAM NAME IS A THREE DIGIT NUMBER.
      CALL INITL (NAME, IPRSW, IALTS, IBED, ITERM,
      ISAVM, ISAVT)
      GO TO (100, 200, 300, 400, 500) IPRSW
C
C*****PROGRAM INITIALIZATION WITHOUT
        PARAMETERS
  100 CALL WORK (10, IERR)
C*****INSURE WORK AREA IS ALLOCATED (IERR IS
        '-' OR '+')
      IF (IERR) 110, 1000, 110
  110 CONTINUE
  CALL MEXIT (ITYPE, IPRSW, IALTS, INTVL)
C
C*****PROGRAM INITIALIZATION WITH PARAMETERS
  200 CONTINUE
C*****WORK STORAGE HAS BEEN ASSIGNED BY
        CALLER
C*****USE 'LDTSK' TO RETRIEVE INITIAL
        PARAMETERS
      CALL MEXIT (ITYPE, IPRSW, IALTS, INTVL)
C
C*****EXTERNAL ENTRY SECTION
  300 CONTINUE
C*****'IALTS' CONTAINS ONE WORD FROM THE
        EXTERNAL CALLER
      GO TO 1000
C
C*****PROGRAM ERROR RECALL SECTION
  400 CONTINUE
C*****'IALTS' CONTAINS THE ERROR CODE
      GO TO 1000
C*************************************************************
  500 CONTINUE
C
C*****PROCESSING COMPLETE EXIT
 1000 CALL MEXIT (0, 0, 0, 0)
      CALL EXIT
      END
```

PROGRAM ORGANIZATION

To enhance response times, core utilization, and disk storage capacity, the MIRU programming system provides multiple entry points to a FORTRAN program. Thus, without being rolled-out (saved) onto disk, a program can be exited and entered at a later time. In-core storage is dynamically available to each program for the preservation of intermediate parameters. While a program is not active (executing), its coreload area is used by other programs. On recall for execution a fresh copy of the program is read from disk and the execution continues from the designated entry point.

This programming system conserves the overhead time that would otherwise be required to save an interrupted coreload on disk and frees a coreload area while a program is inactive. While one task waits for the entry of data from a keyboard, another task program is executing.

Task program

A task program is many program segments connected together under the control of two program switches, the main and timer switches. The switch mechanism is the FORTRAN computed-go-to statement.[8] A program segment begins at one of the statement numbers in the computed-go-to statement and ends with a call to the MIRU executive exit routine. This gives a FORTRAN program multiple entry points.

In the computed-go-to statement, GO TO (100, 200,

300, 400, 500), IPRSW, the numbers are the entry points to the program and "IPRSW" is the main program switch (see Figure 4, MIRU Sample Task Program). By selectively setting "IPRSW" as 1, 2, . . ., N, the program can begin execution at entry points 100, 200, . . . respectively. This provides a powerful but easy mechanism to select the same or different entry point each time the program is executed.

The switches provide the programmer with two independent control paths for each task. The first or main switch is the entry point following data input via a remote terminal or response to an alarm, alert, or message signal.

The second or timer switch is the entry point following a time-out of the task timer. Since the two switches are independent, each may point to a different segment of the program. A program requesting data can regain control after a specified time interval, whether or not data has been entered.

Entry point selection

The MIRU executive initializes the switches at task setup time. At the end of each program segment, the MIRU exit routine sets the switches for the next entry. A task program determines its entry point through a call to the task initialization routine.

A task program begins with the task initialization routine. This routine identifies the task being processed and places the task block address on the MPX level work area, that is, the MPX mechanism for reentrant coding.[7] Subsequent MIRU executive routines reference the task block through the saved address. The initialization routine sets the task active flag and returns the program switches.

The last executable statement of every program segment is a 'CALL MEXIT' (i.e., MIRU executive exit routine). Through this routine, a program communicates with the task block. The user can specify why he is exiting the program (exit type), where he wishes to return (main switch), where to return on a timer recall (timer switch), and the increment of time to remain inactive. The MIRU executive saves this information in the task block, designates the task as inactive, and removes the task block address from the MPX level work area.

The following are task exit types:

0—complete task processing,
1—signal alarm condition,
2—signal alert condition,
3—signal message pending,
4—remain inactive for specified time interval,
5—request data from terminal without erasing display, and
6—request data from terminal and erase display.

Certain main entry points are reserved and defined as follows:

1—task initialized without parameters,
2—task initialized with parameters,
3—task reentered from external source, and
4—task recalled through error condition.

Initial entry without parameters

A task initialized through the terminal has no in-core or disk work storage; and hence, it can have no initial parameters. Working storage can be attained through a request to the MIRU executive. The task program can request up to 255 words of in-core storage and up to 128 sectors of disk storage. This storage space is found by the MIRU executive, which returns its address to the task block. The executive also returns status indicator: Work storage already exists; no work area remains; request successfully filled.

Initial entry with parameters

A task initialized from another task program can be in one of the following states:

- Neither disk nor in-core storage,
- Disk but not in-core storage,
- In-core but not disk storage, or
- Both in-core and disk storage.

A task program starting a new task can transfer data through in-core working storage. The MIRU executive selects a task block, initializes the program switches to entry point 'two', allocates task working storage, moves the data to the storage, and queues the new task for execution.

Using another MIRU executive routine, a program can transfer both in-core and disk working storage to the new task. This routine uses the existing task block for the new task, thus completing the processing by the first task. The parameters contained in in-core and disk working storage are passed to the second task. The program switches are set to entry point 'two'; the new task is queued for execution; and the existing program is exited.

Task working storage cannot be referenced directly. It is accessed indirectly through the work area address stored in the task block. On request, MIRU executive routines load or store values in the working storage. The load and store functions validate the relative

TABLE V—MIRU Abort Messages and Restart Codes

MIRU TASK ABORT MESSAGES

MESSAGE	*MEANING*
'MULT ER ABT'	Multiple error abort.
'WHOOPS BATCH CALL'	System routine called invalidly from BATCH
'WHOOPS TSK AD INV'	No 'CALL INITL' has been made prior to a call to a system routine.
'WHOOPS INV LVL/BT'	No active task found for name input to 'INITL.'
'DAMN DAMN DAMN'	Programmed status dump.

Word	*Definition*
1	Level/Area
2	I (User variable)
3	Terminal/Bed
4	Alternate switch/Main switch
5	Flag/Word count
6	Audit trail switches (Previous entry point)

MIRU ERROR RESTART CODES

4–1	Multiple error recall.
4–2	MPX program restart (FORTRAN error detected).
4–3	Task load/store relative word error.
4–4	Invalid switch in 'MEXIT' call.
4–5	Invalid type in 'MEXIT' call.
4–6	No interval in type 4 'MEXIT' call.
4–7	Invalid word count in 'WORK' call.
4–8	Attempted use of terminal attached to another 'TASK' (e.g., 'CALL TEXT').
4–9	Invalid terminal call in 'MEXIT.'
4–10	Terminal number error in 'ATACH' call.
4–11	Error in code in 'TENTR' call.
4–12	Error in word count in 'TASKP' call.
4–13	Sector count error in 'DWORK' or 'MADDR' call.
4–14	Invalid name in 'TASK,' 'TASKP,' 'TASKT' call.
4–15	Sector count outside reserved area in 'MADDR' call.
4–16	Display number is zero, negative, or greater than number of displays on system.

MIRU EXTERNAL ENTRY CODES

3–1	System reload (Storage protect, op code, or parity error).
3–2	Forced terminal separation (Keyboard reset).

address of the word affected. A task restart occurs if a program attempts to address an invalid core storage location.

External entry

A task reentered through a system reload, through the keyboard reset, or from another task receives a communication code in the alternate (timer) switch. This external entry disrupts the normal flow of the task execution. The MIRU executive saves the former switches; and the task initialization routines returns these to the program. The user can reestablish the flow of execution.

Restart entry

A task restarted through an exception or error condition receives a code in the alternate switch. The MIRU executive leaves control with the program whenever possible. Only the program knows the significance of an exception condition and what corrective action is necessary. On restart, the program can shut down a process or inform the user of the error condition. Only as a program loops in multiple restarts is it forceably aborted. Abort conditions are logged on the system printer (see Table V, MIRU Abort Messages and Restart Codes).

DATA MANAGEMENT

A disk file for each patient's data contains all information needed in real-time. A variety of data types can be saved in the patient file: Raw analog-to-digital values, derived parameters, coded information and narrative data. Information saved in the patient file is used to generate summary displays and for retrospective studies.

MIRU executive routines log each piece of information to a temporary disk file, sort data for individual patient files, and dump disk files to magnetic tape.

When entered into the logging files, information is identified with patient's number, a data type code, and time stamp. A retrieval program collects information from the patient file by code and patient number. Using the code, retrieval routines can extract the requested data without reference to an external source for attributes, e.g., word count. The code also allows logging a group of data, each element identified only by its relative position in the group and the code of the record. This fixed position format is used to log the multiple results of a program without coding each piece of data within the record.

A log request moves the data to a core buffer area. After sixteen requests the core buffer is transferred to the temporary log file on disk. Periodically, a program sorts the temporary disk file (which has data mixed from all patients) for the real-time tape, and individual patient disk file (the patient active file), or the nurse's printer. The tape records are blocked five disk records to one tape record (see Figure 5, Information Logging). When a patient is dismissed from the research unit, a program extracts the data for that patient and gener-

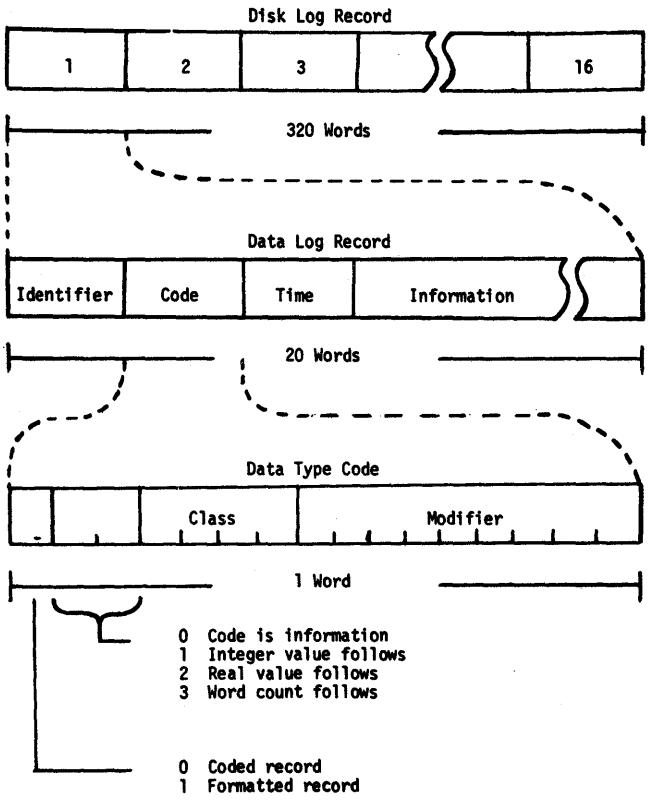


Figure 5—Information logging

ates an individual tape file. This forms the permanent record for retrospective studies.

Analog-to-digital values and large arrays of derived data may be written directly to the real-time tape. In logging data directly to the real-time tape, the user must provide a tape header including patient identification, data type, and time of acquisition. Otherwise, the record will be discarded when the real-time tape is sorted into individual patient files.

CONCLUSION

The research system has been operational since July, 1969, with a single prototype of the medical instrumentation and two keyboard/oscilloscope terminals. The Myocardial Infarction Research Unit opened in October, 1969, so the computer and instrumentation system is being used for research studies from two patient rooms and two laboratories. Instrumentation of an animal laboratory is projected for 1970.

We have not had sufficient experience in a multiple bed research environment to note the strengths and weaknesses of the system. These will be reported at the conference.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the efforts of Mr. Jeary Vogt, now of Shared Medical Systems Corporation, Philadelphia, Pa., in the selection of monitoring equipment and the design of the digital computer interface for the bedside instrumentation. The patient cooperation of Mr. Louis Sheppard of the Department of Surgery in providing consultation, program test time, and proven monitoring programs is appreciated.

REFERENCES

1 H SHUBIN M H WEIL
Efficient monitoring with a digital computer of cardiovascular function in seriously ill patients
Annals of Internal Medicine 65:3 1966

2 H R WARNER R M GARDNER A F TORONTO
Computer-based monitoring of cardiovascular functions in post-operative patients
Supplement II to Circulation 37 1968

3 J J OSBORN J O BEAUMONT S C A RAISON J RUSSELL F GERBODE
Measurement and monitoring of acutely ill patients by digital computer
Surgery 16:6 1968

4 D W CHAAPEL G RASTELLI R WALLACE
On-line computer care of post-operative cardiac patients
Digest of IEEE Computer Group Conference June 1969

5 L C SHEPPARD N T KOUCHOUKOS M A KURTTS J W KIRKLIN
Automatic treatment of critically ill patients following operation
Annals of Surgery 168:4 1968

6 *IBM 1800 data acquisition and control system: Functional characteristics*
IBM Systems Reference Library Form A26-5918

7 *IBM 1800 multiprogramming executive operating system: Programmers guide*
IBM Systems Reference Library Form C26-3720

8 *IBM 1130/1800 basic FORTRAN IV language*
IBM Systems Reference Library Form C26-3715

Linear programming in clinical dental education

by C. E. CRANDELL

University of North Carolina
Chapel Hill, North Carolina

Linear programming techniques have seldom been used in studying educational processes and are just now becoming of interest in health care delivery systems.

The objectives of our pilot research project at the School of Dentistry, University of North Carolina, have been to: (1) demonstrate the feasibility and merits of linear programming for the optimum allocation, scheduling, and utilization of teaching staff and physical facilities in clinical dental education; (2) to demonstrate a practical application of linear programming by introducing the total patient care concept; and (3) to make these data available to other schools.

The introduction of the total patient care concept will provide students with a clinical experience in dental school which will more closely simulate the conditions to be encountered in private practice after graduation. Optimization techniques will permit a comparison of clinical dental education in which students provide comprehensive care to patients as opposed to the traditional point and block systems of fragmented care. Specifically, our pilot program at UNC has sought to demonstrate that the total patient care concept can be introduced without detriment to the quality of clinical teaching or without increasing the cost of clinical dental education. Hopefully, parameters will also be discovered which can be varied in such a way as to reduce costs without detriment to the quality of clinical teaching or to increase the quality of clinical teaching without increasing costs. Either of these worthy objectives would be to the benefit of the student, the school, and the population we serve.

Actually over the past three years, our work has been along the following lines: first, we analyzed the clinical teaching situation in the School of Dentistry and identified those significant factors relative to the effectiveness and efficiency of clinical instruction. These included an analysis of technique procedures that students are required to accomplish, a classification of students in terms of ability, and the time required with reference to these procedures; the rate of utilization of equipment and physical facilities; the rate of utilization of instructors' services and a classification of patients in terms of the complexity of their dental needs. Secondly, we assumed that the relationships between these factors were linear and simulated them conceptionally in a mathematical model of the dental clinic. This information was fed into the computer as representing a typical clinical situation. The functional relationships of these variables have been studied using linear programming techniques. We have identified variables which we feel might represent the dental clinic. The problem for the computer then was to select which set of factors in any of those equations would give us the most efficient operation of the dental clinic. Once we are satisfied that our mathematical model truly represents the clinical situation, we can then experiment with various changes in the operation of the dental clinic within the computer without making any changes in the real world or clinical situation.

The objective function formulation was done by William S. Jewell.

The initial computer experience was with M3LP (SHARE), using 73 equations, 113 variables, 1265 elements, and a density of 15.33. Later, MPS/360 was used on an IBM 360/75, with a preprocessor preparing the inputs.

A run has been made with an expanded matrix of 158 rows, 90 columns, 248 variables, 5990 elements, and a density of 15.28. A feasible solution was found after four iterations; the optimal solution was found on the ninth iteration. The full matrix is estimated to contain 800,000 elements. Consideration is being given to random sampling of inequalities to reduce the size of the matrix to more manageable size.

Early and tentative evaluation of the clinical phase or practical application of this project is probably not of great interest to this audience. However, the dental educators involved believe that linear programming and other operations research techniques will become

useful tools in the delivery of dental care to the American public.

OBJECTIVE FUNCTION OF DENTAL CLINIC MODEL

Minimize

$$C = \sum_{t=1}^{T} \sum_{j=1}^{J} \left[\alpha \sum_{i \epsilon S_j} a_i x_i^t + \beta \sum_{i \epsilon S_j} b_i y_i^t + \phi_j \sum_{i \epsilon S_j} f_i (x_i^t + D_i y_i^t) \right]$$

a = Junior students
b = Senior students
T = Clinical Sessions
J = Clinical Procedures
S_j = Set of procedures in Specialty j
C_j = Number of chairs in Specialty j
$\sum_{i \epsilon S_j}$ = Sum of procedures in Specialty j
P_j = Minimum points for juniors in Specialty j
Q_j = Minimum points for seniors in Specialty j
D_i = 0 for joint procedures; 1 for all others
H_j = Maximum faculty hours in Specialty per quarter
H_j^t = Maximum faculty hours in Specialty j available in time period t.
a_i = Average hours of performance by juniors doing procedure i.
b_i = Average hours of performance by seniors in procedure i.
f_i = Faculty instruction, supervision, checkout, and waiting time in procedure i.
α = Cost per junior student hour
β = Cost per senior student hour
ϕ_j = Cost per faculty hour in Specialty j.
x_i^t = Number, junior students in proc. i in time t.
y_i^t = Number, senior students in proc. i in time t.

Automatic computer recognition and analysis of dental x-ray film*

by DAVID A. LEVINE, HERBERT H. HOPE

State University of New York
Stony Brook, New York

and

MORTIMER L. SHAKUN

USPHS, Tufts University
Boston, Massachusetts

INTRODUCTION

For the past two years work has been in progress at the State University of New York at Stony Brook to develop a hybrid system linking a small digital computer to a programmable flying-spot scanner. This scanner was designed and built to accomplish the task of automatically analyzing x-ray films giving a panoramic view of the teeth, jaw and visually proximate structures, and producing dental diagnostic data as output.

The principal application of this computer-scanner system is to provide a tool for the mass-screening of dental patients and to provide automatic acquisition and storage of dental data.[1] The salient features of this system are:

1. The position of the spot at which the film density is read by the scanner is under program control of the computer. The x-ray film in the scanner is then treated by the computer as a random-access, not fully repeatable memory.
2. The pictorial data is acquired from the film under highly selective program control, involving the use of pattern recognition algorithms. The scanner is directed as a result of these algorithms using sparsely sampled data, to regions of interest. In these regions dental pattern information is sampled selectively, but more densely, to provide the data base for reliable feature recognition.
3. The automatic sequence of region-selection and feature recognition algorithms is not entirely fixed in advance, but is conditionally chosen from a clinical decision-tree under program control, depending on the analysis of the information as acquired.
4. Video display, keyboard, and console-switch control of the system are also provided for man-directed operation. Useful statistical and graphic analysis as well as control procedures are organized in an expandable utility-executive software system with facilities for direct use and user-program linkage. General programs and procedures under development are linked to this utility. Data acquired from scanner operation may also be stored and analyzed off-line.

Although the system has been designed and built to achieve the principal application, it has many features which make it a general tool for other applications.[2,3]

DEVELOPMENT OF THE COMPUTER-SCANNER SYSTEM

The system was designed and achieved in several steps. The steps were to some extent overlapping, but their sequence is as follows. First, a preliminary reconnaissance of available systems was made. No system with the envisioned capability and cost was found at that time. During this reconnaissance most of the algorithm design and program development was initiated in simulated form at the Stony Brook Computing Center. Pictorial data for this simulated system was synthesized as needed and read into a matrix in core memory.

Second, real data, produced from x-ray film scans

* Supported under Contract No. DADA17-67-C-7094 U.S. Army Medical Research and Development Command.

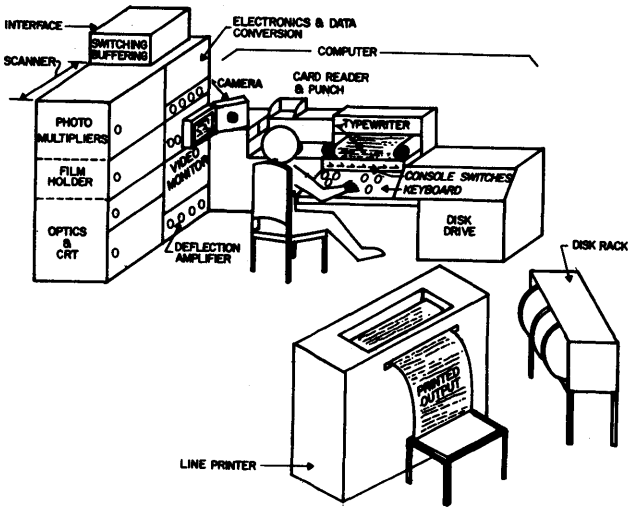


Figure 1—Stony Brook computer-scanner system

with a sequential flying-spot scanner*, and recorded on magnetic tape was used. This data when read into the picture matrix in core memory added realistic contrast levels and noise problems to the simulated system.

Third, a small digital computer, an IBM 1130, was obtained for use as the planned computer component of the hybrid system. The computer chosen has a 16 bit word-length and 8,096 words of core memory. A magnetic disk drive provides storage of over a half-million 16 bit words for relatively fast access to the large volume of programs and data estimated as required. The drive accepts removable disk packs. A card-reader and punch, a line-printer, a keyboard and typewriter and console switches give a useful mix of input-output peripherals. The software, written in FORTRAN, which was used for the simulation at the Computing Center on the IBM 360/67 was then transferred to the 1130 computer. The picture matrix of off-line scanner data which had been stored in core memory of the 360/67 computer was stored on the magnetic disk of the 1130. Submatrices, representing picture "windows" were transferred to the core memory of the 1130 as needed.

Fourth, as algorithm development went forward on the 1130 computer (using data in the off-line mode), a flying-spot scanner was designed with operational characteristics sufficient to meet the estimated requirements of the dental application. The scanner was built and mated to the 1130 computer. After a long and rigorous period of modification, the scanner satisfied the initial requirements of accuracy, speed, repeatability and reliability. At this point, the system was realized as a working basic model of the envisioned hybrid system. (See Figure 1).

THE BASIC FUNCTIONING OF THE SCANNER IN THE SYSTEM

The computer transmits the coordinates, (X, Y), of some selected spot on the x-ray film to two input registers of the computer-scanner interface. The computer next triggers a request to the scanner to return a number, D, representing the gray-level, i.e, the optical density at the requested location, (X, Y). The scanner then responds first, by converting the digital coordinates in the input register to proportional analog form in the scanner. These analog voltages are used to position a spot focussed of light at the requested film location. Next, the amount of light transmitted through the film at the requested spot is measured and converted from an analog voltage to a proportional digital number, D. This number is sent to an output register in the computer-scanner interface. The computer may recover this digital value, D, from the output register any time after the scanner completes its response (See Figure 2).

This request-response cycle may be viewed from the programmer's point of view as operating somewhat as an implementation of a function-subprogram, $D(X, Y)$. Here, X and Y are the arguments and D is the returned function value. In fact, a FORTRAN compatible function subprogram with precisely the form of $D(X, Y)$ was written in assembly language as a basic scanner-interface program. There are some important differences, however, between this hybrid subprogram and a purely digital one, which will be treated below in the section on User Characteristics of the Scanner.

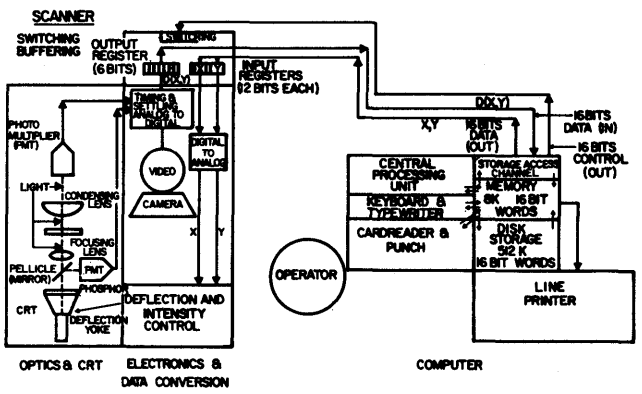


Figure 2—Block schematic of system

* This data was furnished with a sequential scanner at the Albert Einstein Memorial Hospital of Yeshiva University.

SCANNER DATA ACQUISITION

The hybrid function subprogram just described is, for the first model, the basic data acquisition tool of all the programmed algorithms. A single use of this subprogram at a point (X, Y) produces a gray-level value, D, only at a single location on the film.

Half the x-ray film is accessible to the scanner at one time. An electro-mechanical stage, cradling the film-holder, is used to shift to the other half-portion. In each half, any location in a square raster of over four million points (2048 points long by 2048 points wide) may be queried randomly by the hybrid subprogram $D(X, Y)$. Since the physical area of the film thus accessible is about 4.4 inches square, each location of the raster is about .002 inches from its neighbor on the film.

The light transmitted through the film at each location is measured and quantized into sixty-four gray-levels, distributed over an optical density range of zero (clear) to two (dark) in density value.

One view of the scanner, then, is that of a random-access data storage of over four million picture elements, each represented by a six-bit number (sixty-four gray levels). Since the request-response cycle time of the scanner, that is, the speed of the hybrid subprogram is, at this writing, about 200 microseconds, this represents the access time of any data element in the picture storage.

It is evident that with this mass of picture data accessible to the computer, a highly selective procedure for acquiring picture elements is needed. Indeed, the strategy employed leaves the bulk of the data unacquired from the film, utilizing it as a computer storage device.

ALGORITHMS FOR FEATURE LOCATION AND IDENTIFICATION

The features on the panoramic dental x-ray that are noticed, even by the lay observer (see Figure 3), are the dark central area separating the relatively light-colored crowns of the teeth and the tooth-shapes embedded in the background of gray, lacy bone structure. The shape of the roots of the teeth are in some cases clearly discernible and in others often blur into the background a short distance from the tooth-crown.

A computer program has been devised in two phases to operate the scanner in the graphic environment just described producing, in the first phase, a count and identification of teeth present and missing, the identification and location of certain gross pathologies, such as impacted teeth or retained roots, large abcesses, and the location and identification of gross restorations,

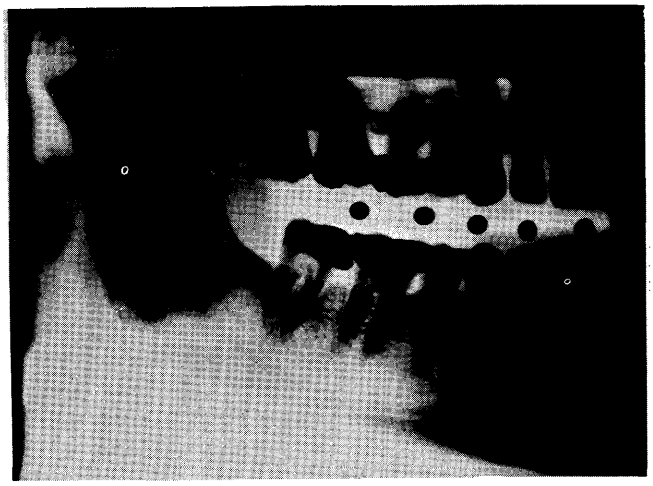


Figure 3—Panoramic dental x-ray picture (positive)

such as bridgework and dummy teeth. In the second phase, a computer algorithm has been devised to examine the density values inside the tooth outlines, sampling the data more heavily and returning a table of features which identifies and locates within the tooth structure pathologies, including caries, and finer restorations such as fillings, root-canal work, and tooth caps.

Heuristic algorithms were developed, many analogizing clinical procedures and decisions in the reading of x-ray film and the recording of results on dental charts and dental-student files[4]. This was followed by programming efforts to design and implement more mathematical algorithms dealing with feature edge-location, and edge-following[5].

A feature of note in Figure 3 is the curved reference block with ball-shaped marks spaced along its length. It is the feature of greatest contrast and is centered between the separated teeth. As a first try, a radio-translucent, plastic bite-block was designed with radio-opaque balls imbedded inside. This bite-block may be placed between the teeth of the patient being x-rayed and produces on the film this unique-reference for the guidance and location of the scanning spot by the computer programs.

Software was written which employs this reference and acquires the coordinates of the edge of the tooth, if present, and the coordinates of the jaw or maxillary bone when a tooth is missing.

UTILITY-EXECUTIVE SOFTWARE SYSTEM

This software system consists of a main program, called UTILY, which accepts commands and param-

eters detailing the command, from the computer console keyboard. UTILY then interprets and executes the command by deploying its various subprograms. Interruption, control, and sequencing of commands is done with the computer console switches for the most part. The system has three main classes of commands.

The first class of commands are those which execute utility subprograms currently compiled into the system. These utility subprograms include those which control the selection of picture sub-matrices, the video-display of the scanner, the production of off-line data from the scanner, the focusing options of the scanner and filtering of data from the scanner. In addition, they process and output on the line-printer and typewriter useful statistical results and pictorial information, including numerical or character picture-matrices, histograms of densities, etc.

The second class of commands are those which provide execution linkage to user-supplied programs. Any user program may thus be executed by the system. A common data area is used to supply parameters to the user-program from the executive and to return parameters to the executive.

The third class of comands are the meta-commands. These include commands which store, retrieve, edit, execute, and provide data for other sequences of commands. In this sense the third class of commands forms the nucleus of a programmable command-language for the hybrid system.

USER CHARACTERISTICS OF THE SCANNER

As mentioned earlier in the paper, the scanner's request-response cycle may be viewed from the programmer's point of view as the implementation of a hybrid subprogram of the form, $D(X, Y)$, where (X, Y) is the argument list of the subprogram giving the location of a picture element and D is the returned value of the function giving the quantized gray-level on the film at location (X, Y). In comparison with a purely digital implementation of $D(X, Y)$ an important difference is evident.

Due to the analog method of controlling spot location, the finite size of the spot of light and its method of generation in the CRT, the method of measuring the transmitted light and quantizing its measurement, etc., the returned value of D is not always the same for successive readings at the same digital location coordinates (X, Y). Initial calibration tests performed on the scanner by repeating cyclically many times a series of density readings on several thousand random film locations show that the modal density value is returned at a location about eighty percent of the time. Some twenty percent of the time a value differing from the mode by one gray level (of a possible 64 levels) is obtained. This amodal response is split about equally between positive and negative deviations. This has been a nearly uniform observation for samples at all 64 density levels and in all film locations.

This scanner response error may be reduced by careful maintenance, calibration and modification of the scanner. A summary of the user characteristics at the time of this writing follows:

SPOT LOCATION:

(1) The spot at the film plane is .006 inches in diameter at the 50 percent intensity circle. This size increases 15 percent at the edge of the raster field.
(2) The raster has 2048 by 2048 picture elements.
(3) Scale at the film plane is 466 picture elements per linear inch.
(4) Raster size at the film plane is 4.4 inches square.
(5) Non-linearity of the pin-cushion type produces a picture element one percent larger at the corner of the raster.
(6) Other non-linearities produce an error less than .2 percent of the raster size.
(7) Location repeatability has an error less than .02 percent of the distance from the electric center of the CRT plus .1 percent of the full raster width.

DENSITY:

(1) Density repeatability for any point has an error of ± 1 density units out of 64. Eighty percent of of the repeated points have the modal quantized value.
(2) The value returned from a location of known density differs from the average of all returns from locations having the same density by not more than ± 2.
(3) Polarity of returned density values: clear is rendered as level 64; dark as level 1.
(4) Conformity of density level spacing to optical density references: the interval between 0. and .30 in optical density contains 13 scanner density levels. The interval between 1.5 and 1.8 in optical density is four scanner density levels. (This non-conformity was produced by optical halation in the CRT faceplate.)

TIMING:

Total response time of the scanner from location query to density value response is 200 microseconds.

SUMMARY

The computer scanner system developed at Stony Brook has as its primary application the automatic scanning of dental x-ray panoramic film operationally used for the mass-examination of dental patients. The application software is designed to produce as output of the system, reliable dental diagnostic data in a form suitable for examination and treatment of large numbers of patients.

The system, however, has been designed with important general-purpose, hybrid graphics processing features which suggest a spectrum of new applications in the bio-medical field and other areas as well. The features which are of general interest include:

1. The present version of the system accepts any transparent film of reasonable size as input.
2. The system has a general utility-executive software system with user-program linkage to help develop applications-software and perform a variety of useful graphics research computations.
3. The system can be viewed as a basic research tool for a spectrum of applications. By using existing programs some work has been done in the examination of films of bone slices* (in conjunction with osteoporosis studies) and the examination of nuclear scintillation-scan film**. Through creation of new software, pattern recognition of a variety of filmed objects is deemed feasible.

---

* In collaboration with E. Pellegrino, R. Biltz, and M. Skolnick of the Stony Brook Health Sciences Center.

** Through the kind cooperation of L. Levy, Long Island Jewish Memorial Hospital.

ACKNOWLEDGMENTS

The project owes a debt of thanks to Dr. David R. Mushabac, who obtained the initial grant, served as project director during the initial stage, and aided in the assembly of the Stony Brook research team.

Much of the software programming, including scanner interface programs, and the UTILY command system is due to the good efforts of Saul Rosenberg, an undergraduate student assistant.

The project's electronic technician, Ambrose Spencer, has been responsible for bringing the scanner into being as a reliable machine.

REFERENCES

1 D A LEVINE
Automatic diagnostic data acquisition for mass screening systems
Proceedings of the Conference on Systems for the Health Sciences Center Stony Brook April 1970

2 A ROSENFELD
Picture processing by computer
Academic Press Inc 1969

3 G S LODWICK
Report of the task force on x-ray image analysis and systems development
U S Public Health Service July 1968

4 M H SKOLNICK D A LEVINE M L SHAKUN
Automated scanning analysis of dental x-rays using the clinical decision tree
Proceedings of the Conference on Computer Applications in Dental Education
Public Health Service San Francisco October 1969

5 M L SHAKUN D A LEVINE H HOPF
Computer image processing of dental panoramic radiographs
Proceedings of the North American Division of the International Association for Dental Research New York March 1970

A translation grammar for ALGOL 68

by VICTOR B. SCHNEIDER

Purdue University
Lafayette, Indiana

INTRODUCTION

In this paper, a translation grammar is presented for a major subset of the ALGOL 68 programming language. This translation is from ALGOL 68 into an intermediate language that was originally designed for an EULER programming system.[9,11] It appears that many of the ALGOL 68 programming facilities, especially the *union* declaration, the use of structures, and the manipulation of arrays with flexible dimension bounds can easily and naturally be expressed using EULER concepts along with some relatively straightforward EULER system procedures. Another advantage of using an EULER intermediate language is that implementations of the Euler system exist on the UNIVAC–1108, IBM–7094, Burroughs–6500, CDC–6500, and doubtless on the IBM–360 line of computers. Therefore, it would require a relatively modest effort for any organization having one of these computers to obtain a working "first version" of ALGOL 68.

The drawbacks of using EULER to implement ALGOL 68 should be mentioned: Since EULER performs run-time type checking, it might be argued that an EULER implementation of ALGOL 68 would run more slowly than necessary. However, it would not be difficult to extend the EULER intermediate language to add operators that do not perform any type checks. At any rate, ALGOL 68 itself demands run-time type-checking because of its use of variables that can store more than one data type and because of the existence of a so-called "conformity operator" that presupposes the existence of a run-time mechanism for keeping track of data types stored within variables. The second objection concerning use of the EULER system is that EULER lists must certainly use computer memory space less efficiently than ALGOL 68 arrays and structures could. This objection seems quite valid. We can only answer by saying that our present translation grammar certainly offers hope of a quick implementation, and that the techniques used in this "toy" translator must certainly have counterparts in a more ambitious code-producing ALGOL 68 compiler. The fact that this smaller version of ALGOL 68 retains the full expressive power of the entire language should not be over looked.

The version of ALGOL 68 to be presented here is simplified in the sense that, for example, not all versions of iterative statements are presented; the mode declaration is only given in its basic form; declarations are ALGOL 60 -style- the only initializations possible are in the priority declaration, the operator declaration, and the mode declaration. These restrictions certainly have no effect on what can be programmed in the language, and any program written in our subset of ALGOL 68 should be immediately transferrable to a standard ALGOL 68 system.

NOTATION FOR A TRANSLATION GRAMMAR

In the translation grammar that follows, an Irons-style notation[1,2,3] is used to specify our subset of ALGOL 68. The following simplified programming language syntax illustrates our notation and introduces some of the basic commands used by our intermediate language:

Grammar 1—A Simple Programming Language

Syntactic Rule		*Rule of Translation*
⟨expr⟩ → ⟨var⟩ = ⟨expr⟩	⇒	⟨var⟩ ⟨expr⟩ *assign*
| ⟨sum⟩	⇒	I

⟨sum⟩ → ⟨sum⟩ + ⟨term⟩	⇒	⟨sum⟩ ⟨term⟩ *add*
| ⟨term⟩	⇒	I
⟨term⟩ → ⟨term⟩ * ⟨factor⟩	⇒	⟨term⟩ ⟨factor⟩ *multiply*
| ⟨factor⟩	⇒	I
⟨factor⟩ → (⟨sum⟩)	⇒	⟨sum⟩
| *at* ⟨var⟩	⇒	⟨var⟩
| ⟨var⟩	⇒	⟨var⟩ *in*
| ⟨var⟩ . (⟨expr–sequence⟩).	⇒	⟨expr–sequence⟩ ⟨var⟩ *in*
⟨var⟩ → ⟨name⟩	⇒	*variable* ⟨name⟩
⟨expr–sequence⟩ → ⟨expr⟩	⇒	I
| ⟨expr–sequence⟩, ⟨expr⟩	⇒	⟨expr–sequence⟩ ⟨expr⟩

Note that the rules of translation above refer to sequences of symbols on the right parts of syntactic rules. In this example, we see that the rules of translation specify how symbols and sequences of symbols in the source language are rearranged and rewritten in the translated language. Where no change at all is indicated in the translation of a particular rule, the symbol "I" appears as a translation rule. As an example of how sequences of symbols are rearranged for translation, the infix addition of

⟨sum⟩ + ⟨term⟩

is translated into the reverse polish sequence of symbols consisting of a "⟨sum⟩" followed by a "⟨term⟩" followed by the intermediate-language command for adding together the values resulting from evaluation of the previous two subexpressions. As in good polish notation, parenthesis are removed from around expressions, and this process is specified by associating the translation rule "⟨sum⟩" with the syntactic rule

⟨factor⟩ → (⟨sum⟩).

The remaining rules having ⟨factor⟩ on the lefthand side are used for translating arithmetic operands into the intermediate language. For example, the syntactic rule

⟨factor⟩ → ⟨var⟩

indicates that operands in arithmetic expressions are variable names, and the translation of a ⟨var⟩ into sequence

⟨var⟩ *in*

indicates that the "*in*" command is used for fetching the value associated with ⟨var⟩ and for storing that value on top of the run-time operand stack of the intermediate-language interpretor.

The other syntactic rule

⟨factor⟩ → *at* ⟨var⟩

reflects the fact the intermediate language permits use of program variables that are pointers to data named by other program variables. Hence, the effect of the "*at*" command of the source language is to suppress the appearance of "*in*" in the translated program after the translated variable name. In this case, a pointer to the data stored in ⟨var⟩ is left on top of the interpretor operand stack at run time. Finally, the rule

⟨var⟩ → ⟨name⟩

means that the names of program variables are translated into the sequence "*variable* ⟨name⟩." Here, the effect of the "*variable*" command is to find a pointer to the data stored in the following name at run time and to place this pointer on top of the run-time operand stack.

The sequence "⟨var⟩.(⟨expr–sequence⟩)." on the right part of the remaining ⟨factor⟩ rule is the definition of a function call. Function calls are translated with the parameters preceding the function name in the translated program. In this way, the function call can be made to look like a reverse polish operator having n operands, with n the number of parameters. A parameterless function call is translated exactly the same way as a program variable. Thus, the sequence

"*variable* ⟨name⟩ *in*"

in a translated program serves both to fetch data and to initiate a call on a function, depending on the ⟨name⟩ involved. This calling sequence will be referred to in the full grammar that follows.

We have attempted to make our translation grammar for program variables. Hence, the effect of the "*at*" command of the source language is to suppress the appearance of "*in*" in the translated program after the translated variable name. In this case, a pointer to the data stored in ⟨var⟩ is left on top of the interpretor operand stack at run time. Finally, the rule

⟨var⟩ → ⟨name⟩

means that the names of program variables are translated into the sequence "*variable* ⟨name⟩." Here, the effect of the "*variable*" command is to find a pointer to the data stored in the following name at run time and to place this pointer on top of the run-time operand stack.

The sequence "⟨var⟩.(⟨expr–sequence⟩)." on the right part of the remaining ⟨factor⟩ rule is the definition of a function call. Function calls are translated with the parameters preceding the function name in the translated program. In this way, the function call can be made to look like a reverse polish operator having n operands, with n the number of parameters. A parameterless function call is translated exactly the same way as a program variable. Thus, the sequence

"*variable* ⟨name⟩ *in*"

in a translated program serves both to fetch data and to initiate a call on a function, depending on the ⟨name⟩ involved. This calling sequence will be referred to in the full grammar that follows.

We have attempted to make our translation grammar for ALGOL 68 as self-contained as possible. Thus, each translation rule is followed by an explanation of what effects the intermediate language commands are producing. It should be noted that the larger grammar uses, e.g., the symbol "=" in place of the "*assign*" command of our small example, and, in general translates as many source symbols as possible into similar commands of the intermediate language. A full description of the intermediate language can be found in Schneider.[9,10]

PROGRAM STRUCTURE IN ALGOL 68

Since many programming features in ALGOL 68 are defined in terms of ALGOL 68 constructs, ALGOL 68 programs must contain an outer block in which these features are defined. Thus, an ALGOL 68 program consists of an inner block which is the "particular program" of some programmer and an outer block containing library subroutines and standard definitions. For the moment, we will concentrate on particular programs and treat the outer block as though it were an implementation-dependent set of control cards necessary to the running of ALGOL 68 programs. Thus, we can write syntactic rule 1:

⟨program⟩ → (⟨standard prelude⟩ ⟨block⟩; exit:
⟨standard postlude⟩) (1)

Here, the sequence "exit:" is a label definition that is global to everything in the particular program ⟨block⟩. Thus, a particular program can always be terminated by a statement such as

go to exit;

The left and right parentheses used to surround the outer block of a program are essentially interchangeable with *begin* and *end*, which are also symbols in ALGOL 68:

⟨block⟩ → *begin* ⟨clause⟩ *end* ⇒ ⟨clause⟩
| (⟨clause⟩) ⇒ ⟨clause⟩ (2)

Except in a few cases, ALGOL 68 treats blocks as though they were expressions that have values. Thus, the statement

$$a := bx(c + d);$$

means the same thing in ALGOL 60 as in ALGOL 68: The value obtained from adding c to d is multiplied by the value of b and stored in the location denoted by a. The ALGOL 68 statement

$$a := b \times \textit{begin real } y; y := 1; y + a \textit{ end}$$

means that the value obtained from adding y to a in the block will be multiplied by the value of b and stored in the location denoted by a. Note that here the sequence

"$y + a$"

is *treated as a statement* of the language that yields a *value.*

DECLARATIONS AND DATA TYPES

Variables declared in an ALGOL 68 block are local to that block in essentially the same fashion as in ALGOL 60—only, in ALGOL 68, these declared variables are said to be "protected" instead of local. As near as I can determine, protection of variables involves tagging them with a higher block number than the variables of their outer block:

⟨clause⟩ → ⟨statementsequence⟩ ⇒ I
| ⟨declarations⟩; ⟨statementsequence⟩
⇒ $BEGIN ⟨declarations⟩; ⟨statementsequence⟩
$END (3)

In this translation rule, the $BEGIN command of the intermediate language increments the block number count, and $END decrements that count. In addition, the $END command initiates a "garbage collection" procedure that dismantles arrays and data structures constructed within its block. Since more than one declaration can appear in a block, we have the addi-

tional rule:

$$\begin{aligned}\langle\text{declarations}\rangle &\rightarrow \langle\text{declaration}\rangle \quad \Rightarrow \text{I}\\ &\mid \langle\text{delcarations}\rangle;\ \langle\text{declaration}\rangle \Rightarrow \text{I}\end{aligned} \tag{4}$$

There are four primitive data types used in ALGOL 68 declarations. In addition, data types can be constructed that consist of structures containing the primitive types together with other invented data types. Rules (5) list the primitive types, together with the ⟨indicant⟩ that has the same effect as a data type declaration:

$$\begin{aligned}\langle\text{declarator}\rangle &\rightarrow \textit{real} \quad \Rightarrow \$\text{NUMBR 0}\\ &\mid \textit{long real} \quad \Rightarrow \$\text{NUMBR 0}\\ &\mid \textit{int} \quad \Rightarrow \$\text{NUMBR 0}\\ &\mid \textit{long int} \quad \Rightarrow \$\text{NUMBR 0}\\ &\mid \textit{bool} \quad \Rightarrow \text{FALSE}\\ &\mid \textit{char} \quad \Rightarrow .*\_\\ &\mid \langle\text{indicant}\rangle \Rightarrow \$\text{VARBL } \langle\text{indicant}\rangle \$\text{IN}\end{aligned} \tag{5}$$

Note that the "long" prefix indicates an extra multiple of arithmetic precision. A primitive data type has no translation, since it provides type information to the compiler. By contrast, an ⟨indicant⟩ is some data type designation invented by the programmer in the course of writing his program. Naturally, a given ⟨indicant⟩ can only have one definition attached to it at any given point in a program:

$$\langle\text{indicant}\rangle \rightarrow \langle\text{name}\rangle \Rightarrow \text{I} \tag{6}$$

The basic data types can be prefixed with what are essentially attributes and then concatenated together into regular expressions that yield information about, e.g., whether a variable is a reference to the data stored in another variable or a procedure or some structural combination of references, procedures, invented data types, etc.

$$\begin{aligned}\langle\text{indication}\rangle &\rightarrow \langle\text{declarator}\rangle && \Rightarrow \langle\text{declarator}\rangle\\ &\mid \textit{ref}\ \langle\text{indication}\rangle && \Rightarrow \$\text{VARBL } \$\text{REF}\\ &\mid \textit{ref}\ [\ ]\ \langle\text{indication}\rangle && \Rightarrow \$\text{VARBL } \$\text{REF}\\ &\mid \textit{ref}\ [\langle\text{boundslist}\rangle]\ \langle\text{indication}\rangle && \Rightarrow \$\text{VARBL } \$\text{REF}\\ &\mid \textit{proc} && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ (\langle\text{typelist}\rangle) && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ [\ ]\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ [\langle\text{boundslist}\rangle]\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ (\langle\text{typelist}\rangle)\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ (\langle\text{typelist}\rangle)\ [\ ]\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{proc}\ (\langle\text{typelist}\rangle)\ [\langle\text{boundslist}\rangle]\ \langle\text{indication}\rangle && \Rightarrow .\$2\$.\\ &\mid \textit{struct}\ (\langle\text{typepack}\rangle) && \Rightarrow .\ (\langle\text{typepack}\rangle)\ .\\ &\mid \textit{union}\ (\langle\text{declaratorpack}\rangle) && \Rightarrow \$\text{UNDEF}\end{aligned} \tag{7}$$

It can be seen in rules (7) that "*ref*" is a prefix attribute indicating that the variable being declared is a pointer. Thus, the declaration

$$\textit{ref}\ [1{:}50]\ \textit{real}\ a;$$

means that the variable "*a*" can contain a pointer to an array of fifty floating-point numbers. The declaration

$$\textit{proc}\ (\textit{real}, \textit{int})\ [1{:}5]\ \textit{int}\ u;$$

means that the variable "*u*" is a procedure whose two parameters are respectively "real" and "integer," and that the procedure returns a value consisting of a five-element integer array.

In our translation scheme, declarations are used to initialize the data types of variables and to enter these variables on the run-time name table. Thus, in rules 5, all numbers are initialized to zero by the "$NUMBR 0" sequence; logical variables are initialized by the "$FALSE" datum; characters are initialized to a blank by the ".*-" sequence; and invented data types are called onto the operand stack by the "$VARBL ⟨indicant⟩ $IN" sequence that fetches the data definition associated with the invented ⟨indicant⟩. In rules (7), we find that all variables whose declarations are prefixed with *ref* are initialized to point to an undefined system variable "$REF." The sequence "$VARBL $REF" brings the pointer to $REF onto the run-time operand stack. All variables whose declarations are prefixed with "*proc*" are set equal to the empty procedure definition ".2.". Finally, the "*struct*" declaration constructs a list of the typed elements within it and sets the declared variables equal to a copy of that list. Since all variables in our system can actually store any data type or structure, any "*union*" declaration simply initializes the declared variables to the value "$UNDEF" (meaning "undefined").

The mechanism for constructing a prototype list in a *struct* declaration is given in (8):

```
⟨typepack⟩ → ⟨type⟩ ⟨name⟩
           ⇒ ⟨type⟩                                  (8)
⟨typepack⟩ → ⟨typepack⟩, ⟨type⟩ ⟨name⟩
           ⇒ ⟨typepack⟩, ⟨type⟩
```

Because the translator must retain subscripting information for declared structures, it is assumed that a copy of each structure declaration will be stored by the translator, and that the translator can discover situations in which one structure is an element of another structure.

Rules (9) tell us that a ⟨type⟩ can define arrays or simple variables:

```
⟨type⟩ → ⟨indication⟩                  ⇒ I
       | [ ] ⟨indication⟩              ⇒ . (⟨indication⟩) .
       | [⟨boundslist⟩] ⟨indication⟩
       ⇒ . (⟨boundslist⟩) . ⟨indication⟩ $VARBL $ARRAY $IN          (9)
```

In rules (9), simple variable declarations are left unchanged, but array declarations cause lists to be constructed. In the case of empty array brackets "[ ]," a one element list appears in the translation. To change the size of such an array, the programmer must assign sets of values to appropriate elements of the array. Array declarations having nonempty boundslists cause the system procedure "$ARRAY" to be called, and this procedure constructs an appropriately dimensioned list of lists, each element of which contains a datum given by the translation of ⟨indication⟩. In the typedeclaration of rules (10), the translations in rules (9) are copied by the system procedure "$COPY" that initializes and declares variables at run time. Because $COPY returns as its value a copy of its first parameter, a chain of calls on $COPY allows the system to handle declarations such as

[1:10, 1:200] *real x, y, z;*

in which three separate two-dimensional arrays must be constructed.

```
⟨typedecl⟩ → ⟨type⟩ ⟨name⟩
    ⇒ ⟨type⟩ $NEW ⟨name⟩ $VARBL ⟨name⟩ $VARBL $COPY $IN
⟨typedecl⟩ → ⟨typedecl⟩, ⟨name⟩
    ⇒ ⟨typedecl⟩ $NEW ⟨name⟩ $VARBL ⟨name⟩ $VARBL $COPY $IN          (10)
```

Procedures "$COPY" and "$ARRAY" are documented in Appendix 1.

In the translation scheme above, the system subroutine $COPY is set into action by the calling sequence "$VARBL $COPY $IN." The two parameters of $COPY are stored in sequence on the run-time operand stack. Its first parameter is a list or a value. Following this parameter, the $NEW command allocates a space on the run-time nametable for the declared ⟨name⟩, and then a pointer to that ⟨name⟩ is called onto the operand stack by the sequence "$VARBL ⟨name⟩." Thus, both parameters are loaded onto the operand stack before the $COPY routine is called. $COPY is programmed to return a copy of its first parameter to the top of the operand stack as its value. As a side effect, storage is allocated for ⟨name⟩. Thus, the second rule of (10) works in the same way as was just described, only here, the first parameter of $COPY is supplied by the previous call on $COPY that was made by a ⟨typedec⟩.

To complete our description at this point, we give the translations of ⟨boundslist⟩ and ⟨declaratorpack⟩:

```
⟨boundslist⟩ → ⟨bounds⟩                  ⇒ I
             | ⟨boundslist⟩, ⟨bounds⟩ ⇒ I                (11)

⟨bounds⟩ → ⟨sum⟩(1) : ⟨sum⟩(2)  ⇒ ⟨sum⟩(1) ; ⟨sum⟩(2)
         | ⟨sum⟩:flex            ⇒ ⟨sum⟩
         | flex:⟨sum⟩            ⇒ ⟨sum⟩              (12)
         | flex:flex             ⇒ $UNDEF

⟨declaratorpack⟩ → ⟨uniteddeclarator⟩ ⇒ I
                 | ⟨declaratorpack⟩, ⟨uniteddeclarator⟩
                  ⇒ ⟨declaratorpack⟩ ⟨uniteddeclarator⟩      (13)

⟨uniteddeclarator⟩ → ⟨declarator⟩ ⇒ –
                   | [ ] ⟨declarator⟩      ⇒ –          (14)
                   | union (⟨declaratorpack⟩) ⇒ –
```

In the translation of array bounds in ALGOL 68, one has to take into account the possibility of undefined array limits indicated by the "[ ]" sequence or by

the following translation rules:

⟨clausesequence⟩ → ⟨clause⟩ ⇒ .$ ⌊⌋⟨clause⟩$.↑

| ⟨clausesequence⟩ → ⟨clausesequence⟩, ⟨clause⟩

⇒ ⟨clausesequence⟩,.$ ⌊⌋⟨clause⟩$.↑ (17)

A typical procedure definition in the translation of ⟨clausesequence⟩ would look like the sequence

".$ ⌊⌋⟨clause⟩$.↑"

Here, the ".$" command tells the run-time system that what follows is a procedure definition. It therefore leaves as its value the value of the translated program *location counter* that begins the translated ⟨clause⟩. Since the procedure is *not* activated when the procedure definition is assigned to some variable, the ".$" command is followed by a jump to the code directly following the procedure definition. The "$." command is part of the code in the procedure definition. The "$." is executed as a return jump command that looks up a return label on a table of return jumps and transfers control back to that point in the program where the procedure was called.

Finally, for our particular choice of intermediate language, subscripting is accomplished by the ")" command. This ")" command assumes that the top-most operand of the run-time operand stack is an integer number, and that the next-to-top operand is a reference to a list cell. Thus, we have the translated sequence

⟨subscriptvar⟩ ⟨subscripts⟩)

given in rules (16), and the following rules for translation of ⟨subscripts⟩:

⟨subscripts⟩ → ⟨sum⟩ ⇒ I

| ⟨subscripts⟩, ⟨sum⟩ ⇒ ⟨subscripts⟩) ⟨sum⟩ (18)

To complete this description of subscripting, we introduce the next layer of rules above rules (17) in the system of precedence:

⟨selection⟩ → ⟨subscriptvar⟩ ⇒ I

| ⟨name⟩ *of* ⟨selection⟩ (19)

⇒ ⟨selection⟩ ɜ [⟨name⟩]

Thus, in our syntax, the numerical subscripting of (16) takes precedence over the logical subscripting of (19) because of the ordering of rules. When a variable is logically subscripted as described in (19), the translator provides a numerical subscript to the translated program, and this numerical subscript corresponds to the position in its own structure of the logical subscript that is used. The notation

ɜ [⟨name⟩]) (20)

in rules (20) represents the translator-supplied subscript number followed by the subscripting command. This subscripting capability of course implies that the translator must itself keep track of structures and pointers from one element of a structure to another structure. In this way, the translator must have a list processing capability for tracing lists and sublists to any desired depth of nesting.

DATA CONSTANTS AND PROCEDURE DEFINITIONS IN ALGOL 68

At this point, it is convenient to introduce the data types used in the language. Data of type *char* (character) is denoted by the following syntax:

⟨charprim⟩ → "⟨alphameric⟩" ⇒ .*⟨alphameric⟩ (21)

The sequence ".*⟨alphameric⟩" stores that symbol on top of the run-time operand stack. Here, ⟨alphameric⟩ is whatever set of symbols are available for a particular computer. By convention, a quote symbol (") is represented by a pair of quotes (" ") in the language. Thus, the assignment

v: = " " " ";

stores a single quote in character variable *v*.

Data of type *bool* (logical) is denoted by the following syntax:

⟨logicalprim⟩ → *true* ⇒ $TRUE

| *false* ⇒ $FALSE (22)

The $TRUE ($FALSE) command stores the internal representation of logical truth (or falsety) on top of the run-time operand stack.

Although ALGOL 68 allows real and integer numbers, as well as multiple precision versions of numbers, we have for simplicity translated all numbers into single-precision floating point:

⟨number⟩ → ⟨integer⟩ ⇒ $NUMBRɜ[⟨integer⟩]

| ⟨integer⟩ . ⟨integer⟩

⇒ $NUMBRɜ[⟨integer⟩ . ⟨integer⟩] (23)

In both cases of rules (23), the command "$NUMBR" is followed by the internal floating point representation of the appropriate character strings. "$NUMBR" serves to place the following translated word on top of the run-time operand stack.

For convenience in initializing small arrays, ALGOL 68 allows structures similar in appearance to lists. So, we will call them lists, instead of using the ALGOL 68

"flex" in place of a definite upper or lower bound. As handled in the translation scheme above, all arrays translate into lists. Hence, a lower index bound of 1 always exists for each dimension of an array, whether or not the programmer asks for that lower bound. In addition, arrays with no bounds specified for some dimension contain one element undefined sublists for that dimension.

Translation of variables declared to be of type *union* is simple and direct, since our run-time system treats all variables as though they were capable of storing any legal data type. The translator merely keeps a record of which variables are of particular data type.

VARIABLES AND SUBSCRIPTING IN ALGOL 68

From the syntax in the preceding section, we see that the declaration

struct (*real x*, [1:10, 1:5] *int y*)*z*;

is legal in ALGOL 68. This declaration causes *z* to refer to a two-element structure of values whose second element is a two-dimensional array that stores fifty integers. In order to store values into *z* or extract values from *z*, these elements of *z* are referred to by name; e.g.,

x of z := (*y of z*) [8, 3];

In this statement, the forty-third integer in the second element of *z* is converted to a real number and stored into the first element of *z*. Thus, we have two methods for subscripting variables in ALGOL 68, and these methods can be used separately or in combination.

Another feature of variable usage in ALGOL 68 is that variables can be "selected" before being subscripted. Thus, the statement

a := *if gl then b else c fi* [5, 3];

is valid in ALGOL 68 if *a*, *b*, *c* and *gl* are appropriately declared, and the subscripts "[5, 3]" are within the bounds of *b* and *c*. Another, essentially similar, usage is one in which a case statement (similar to the case statement in ALGOL *W*) is used for selecting a variable:

if gl then d else e fi := *case i in a, b, c esac* [2, 4];

This use of conditional expressions for selecting variables in ALGOL 68 leads to a translation difficulty in which the translator is unable to decide whether it is translating a chain of variable references or expressions yielding values because either interpretation is correct for a conditional expression. To get around this difficulty, we have decided to write our translation grammer so as to treat every expression as though it were a variable reference until the last possible minute. Thus, we obtain the following strange-looking syntax of subscripted variables:

$$\begin{aligned} \langle \text{name} \rangle &\rightarrow \langle \text{letter} \rangle &&\Rightarrow \text{I} \\ &\mid \langle \text{name} \rangle \langle \text{letter} \rangle &&\Rightarrow \text{I} \\ &\mid \langle \text{name} \rangle \langle \text{digit} \rangle &&\Rightarrow \text{I} \end{aligned} \qquad (15)$$

In rules (15), a table lookup mechanism informs the translator that some ⟨name⟩ is actually a declared variable. Then, rules (16) are applied.

$$\begin{aligned} &\langle \text{subscriptvar} \rangle \rightarrow \langle \text{name} \rangle && \Rightarrow \$\text{VARBL } \langle \text{name} \rangle \\ &\quad \mid \langle \text{block} \rangle && \Rightarrow \text{I} \\ &\quad \mid \textit{if } \langle \text{clause} \rangle^{(1)} \textit{ then } \langle \text{clause} \rangle^{(2)} \textit{ else } \langle \text{clause} \rangle^{(3)} \textit{ fi} && \Rightarrow \langle \text{clause} \rangle^{(1)} \$\text{IF} \llcorner\!\lrcorner \langle \text{clause} \rangle^{(2)} \$\text{THEN} \llcorner\!\lrcorner\!\uparrow \langle \text{clause} \rangle^{(3)} \uparrow \\ &\quad \mid \textit{case } \langle \text{clause} \rangle \textit{ in } \langle \text{clausesequence} \rangle \textit{ esac} && \Rightarrow \langle \text{clause} \rangle \,.\, (.\$ \langle \text{clausesequence} \rangle \$.\rangle \,.\, \$\text{VARBL } \$\text{CASE } \$\text{IN} \\ &\quad \mid \langle \text{subscriptvar} \rangle [\langle \text{subscripts} \rangle] && \Rightarrow \langle \text{subscriptvar} \rangle \langle \text{subscripts} \rangle] \end{aligned} \qquad (16)$$

In rules (16) above, it should be explained that the translator must have an operand stack that permits it to keep track of data types associated with variables and expressions. With this operand stack, the translator will only allow subscripting to follow an expression whose value is a variable reference. In translating the conditional statement of rules (16), the translator also uses a stacking mechanism to supply program labels to the conditional branch command "$IF" and the unconditional branch command "$THEN." These labels are indicated schematically in the rules by the "$\llcorner\!\lrcorner\!\uparrow$" notation. Finally, a case statement is translated into a two-parameter procedure call on the system procedure "$CASE." The first parameter is an expression that has a subscript as its value, and the second parameter is a run-time list of parameterless procedure definitions that is to be subscripted by the first parameter in the process of calling one of the list elements.

This list of procedure definitions is constructed by

name for them:

⟨listprim⟩ → (⟨clause⟩, ⟨listend⟩
⇒ . (⟨clause⟩, ⟨listend⟩
⟨listend⟩ → ⟨clause⟩) ⇒ ⟨clause⟩) .
| ⟨clause⟩, ⟨listend⟩ ⇒ ⟨clause⟩, ⟨listend⟩ (24)

Here, if we use the declaration "[1:2] *real x*" and follow this by the assignment "*x* := (1.0, 3.5)"; we can see how the lists are used. Thus, these list primaries translate directly into our own notation, with the exception that lists of zero or one elements cannot be *represented* using list notation. This is to avoid an ambuiguity, e.g., in which the sequence "(1)" is interpreted as being both a ⟨block⟩ and a ⟨listprim⟩.

In an analogous vein, ALGOL 68 has provisions for representing strings of symbols:

⟨stringprim⟩ → "⟨alphameric⟩ ⟨stringend⟩
⇒ . (. *⟨alphameric⟩, ⟨stringend⟩
⟨stringend⟩ → ⟨alphameric⟩" ⇒ . * ⟨alphameric⟩) .
|⟨alphameric⟩ ⟨stringend⟩⇒ . *⟨alphameric⟩, ⟨stringend⟩ (25)

Thus, a string translates into an array of characters in the same fashion as in ALGOL 68. Here again, strings of length zero and one cannot be directly represented to prevent ambiguity. As we will see, it is possible to invent a data type called *string* whose variables all consist of character arrays with flexible upper bounds.

Finally, we include procedure definitions in this section because procedure definitions can be assigned to variables in the same way as other data constants. We have chosen to use an older version of the ALGOL 68 procedure definitions, because the keyword "*expr*" used in these older versions is subject to fewer usages in the language than the ":" that replaces *expr* in the syntax. We have also chosen to force the programmer to be explicit about his choice of procedure definitions in our version. Thus, the assignment statement

*v*1 := *if g*1 *then expr v*2 *else expr v*3 *fi*;

(Assign procedure definition *v*2 to *v*1 if *g*1 is true, otherwise assign procedure definition *v*3.) is legal in our version of ALGOL 68 as well as the "full" version of the language. But we do *not* allow the alternate assignment statement:

*v*1 := *if g*1 *then v*2 *else v*3 *fi*;

even though this is also legal in ALGOL 68 and is equivalent to the first statement. We thus have the syntax of procedure definitions in rules (26):

⟨procdef⟩ → *expr* ⟨assignment⟩ ⇒ .$⌴⌴⟨assignment⟩ $.↑
| ⟨indication⟩ *expr* ⟨assignment⟩ ⇒ .$⌴⌴⟨assignment⟩ $.↑
| (⟨p. list⟩) *expr* ⟨assignment ⇒ .$⌴⌴⟨p. list⟩ ⟨assignment⟩ $.↑
| (⟨p. list⟩) ⟨indication⟩ *expr* ⟨assignment⟩ ⇒ .$⌴⌴⟨p. list⟩ ⟨assignment⟩ $.↑ (26)

These procedure definitions are of the same form as described in rules (18). However, here the procedure definition symbols ".$" and "$." surround what will be seen to be single statements. In order for a procedure to be permitted to return a value in ALGOL 68, the type of its returned value must be indicated in the procedure definition. Thus, all the procedure definitions having an ⟨indication⟩ before the *expr* keyword behave like ALGOL 60 or FORTRAN functions, while the remaining definitions are similar to procedured in those languages. Finally, we have the translation syntax of the parameter list:

⟨p. list⟩ → ⟨type⟩ ⟨name⟩ ⇒ $FORMA ⟨name⟩
| ⟨p. list⟩, ⟨name⟩ ⇒ $FORMA ⟨name⟩ ⟨p. list⟩
| ⟨p. list⟩, ⟨type⟩ ⟨name⟩ ⇒ $FORMA ⟨name⟩ ⟨p. list⟩ (27)

Here, the formal parameters are translated into a reverse sequence. The command "$FORMA" has the effect of entering the following ⟨name⟩ of a parameter onto the run-time name table. $FORMA then fetches the topmost operand on the runtime operand stack, and stores its value and type into the newly declared parameter ⟨name⟩. Because of this method for passing parameters to procedures, the old call-by-name and call-by-value conventions in ALGOL 60 procedures have been abandoned in favor of allowing the programmer to pass procedure definitions, pointers to variables, and actual values into his procedure calls. As an illustration, if *v*1 is a reference variable, *v*2 is an integer variable, and *v*3 is an appropriate procedure

name, then the procedure call

$$v3\ (v1,\ \textit{int expr}\ v2,\ 5)$$

passes as parameters to the procedure definition of $v3$ a pointer stored in $v1$, a procedure definition whose body "calls" $v2$, and an integer value. As we will see, the procedure calling mechanism puts these three parameters onto the operand stack in the right sequence for allowing the $FORMA commands to pick off their values for the formal parameters.

PROCEDURE CALLS AND EXPRESSION PRIMARIES

In a number of preceding rules, we implicitly assumed the existence of a procedure calling mechanism in our intermediate language. This mechanism works as follows: A typical procedure call is of the form

⟨value 1⟩ ⟨value 2⟩ · · · ⟨value n⟩ $VARBL ⟨name⟩ $IN.

The n values are stored in sequence on the run-time operand stack, and then the command sequence "$VARBL ⟨name⟩ $IN" passes control to the procedure. This procedure is assumed to have n formal parameters, and therefore n "$FORMA ⟨name⟩" sequences at the beginning. These "$FORMA ⟨name⟩" sequences pick off the values stored on the operand stack in the reverse of the sequence in which they were stored. Hence, any procedure calling system must have a mechanism for matching actual and formal parameters. When we discuss the extendible operations feature of ALGOL 68, we will justify the necessity for our use of this procedure calling command sequence.

The mechanism needed for matching actual and formal parameters is further complicated by the legality of ALGOL 68 procedure calls such as in the following example:

$$\textit{if}\ g1\ \textit{then}\ p1\ \textit{else}\ p2\ \textit{fi}\ [3,\ 9]\ (4,\ \textit{real}\ \text{expr}\ p3)\,;$$

Here, one of two procedure arrays is chosen by a conditional statement. Then, the chosen procedure array is subscripted, and finally, the actual parameters are supplied when that procedure element is called. To explain our method for translating such ALGOL 68 procedure calls, we give the following syntax of procedure calls:

⟨procall⟩ → ⟨subscriptvar⟩ (⟨a. p. list⟩)
⇒ ⟨subscriptvar⟩ . (⟨a. p. list⟩) . $VARBL $XEQUT $IN (28)

We see that a program procedure call with parameters is translated into a call on a system procedure named "$XEQUT." This is accomplished by transforming the program procedure call into two parameters. The first parameter is a (possibly subscripted or selected) pointer to where the procedure call is stored on the run-time name table. The second parameter is a link to a list constructed from the actual parameter list of the program by rules (28) and (29):

⟨a. p. list⟩ → ⟨assignment⟩ ⇒ I
| ⟨a. p. list⟩, ⟨assignment⟩ ⇒ I (29)

The listing of the $XEQUT procedure can be found in Appendix 1.

A procedure call without parameters can be treated syntactically like an ordinary value primary of the language. This is because the sequence

⟨selection⟩ $IN

is sufficient to execute procedures as well as to bring values referenced by ⟨selection⟩ pointers to the operand stack.

We can thus begin to give a syntax for expression primaries in ALGOL 68:

⟨prim⟩ → ⟨procall⟩ ⇒ I
| ⟨procdef⟩ ⇒ I
| ⟨stringprim⟩ ⇒ I
| ⟨listprim⟩ ⇒ I
| ⟨number⟩ ⇒ I
| ⟨logicalprim⟩ ⇒ I
| ⟨charprim⟩ ⇒ I
| ⟨selection⟩ ⇒ ⟨selection⟩ $IN
| ⟨referenceprim⟩ ⇒ I
| *val* ⟨prim⟩ ⇒ ⟨prim⟩ $IN (30)

In rules (30) above, the $IN command is supplied by the translator to fetch values in two instances. For the first instance to apply, the translator program must determine by inspection of its operand stack that ⟨selection⟩ appears after the left part of an assignment statement such that the left part is not of type reference. This is because the assignment statement,

$$a := b;$$

where "a" is a variable of type reference, is legal in ALGOL 68. Thus, "b" must be treated as a ⟨referenceprim⟩, and the translator must determine by context (using its own operand stack) whether or not some ⟨selection⟩ is a ⟨referenceprim⟩ as in rule (31) or a value-bearing ⟨prim⟩ as in rules (30):

⟨referenceprim⟩ → ⟨selection⟩ ⇒ I (31)

Of course, the sequence

val ⟨prim⟩

means that the programmer explicitly desires to fetch a value to which a ⟨referenceprim⟩ refers. So as to spare the translator from the necessity of trying to discover how many layers of pointers must be traced through in order to fetch a value, we will require the use of *val* whenever a ⟨referenceprim⟩ is to be "depressed" to a value in the middle of an expression.

EXPRESSIONS AND THE PRECEDENCE OF OPERATORS

In ALGOL 68, there are ten levels of operator precedence. Corresponding to each level is a set of standard operators for that level. These operators can be redefined by the programmer, who may change their precedences, introduce new procedures that describe their actions, and introduce new operators to suit his convenience. The syntax for such redefinable operations must take into account this new facility:

⟨unary⟩ → ⟨op 10⟩ ⟨prim⟩ ⇒ ⟨prim⟩ $VARBL ⟨op 10⟩ $IN
| *go to* ⟨prim⟩ ⇒ ⟨prim⟩ $GOTO
⟨complex⟩ → ⟨unary⟩ ⇒ I
| ⟨complex⟩ ⟨op 9⟩ ⟨unary⟩ ⇒ ⟨complex⟩ ⟨unary⟩ $VARBL ⟨op 9⟩ $IN
⟨exponent⟩ → ⟨complex⟩ ⇒ I
| ⟨exponent⟩ ⟨op 8⟩ ⟨complex⟩ ⇒ ⟨exponent⟩ ⟨complex⟩ $VARBL ⟨op 8⟩ $IN
⟨product⟩ → ⟨exponent⟩ ⇒ I
| ⟨product⟩ ⟨op 7⟩ ⟨exponent⟩ ⇒ ⟨product⟩ ⟨exponent⟩ $VARBL ⟨op 7⟩ $IN
⟨sum⟩ → ⟨product⟩ ⇒ I
| ⟨sum⟩ ⟨op 6⟩ ⟨product⟩ ⇒ ⟨sum⟩ ⟨product⟩ $VARBL ⟨op 6⟩ $IN
⟨inequality⟩ → ⟨sum⟩ ⇒ I
| ⟨sum⟩$^{(1)}$ ⟨op 5⟩ ⟨sum⟩$^{(2)}$ ⇒ ⟨sum⟩$^{(1)}$ ⟨sum⟩$^{(2)}$ $VARBL ⟨op 5⟩ $IN
⟨confrontation⟩ → ⟨inequality⟩ ⇒ I
| ⟨inequality⟩$^{(1)}$ ⟨op 4⟩ ⟨inequality⟩$^{(2)}$ ⇒ ⟨inequality⟩$^{(1)}$ ⟨inequality⟩$^{(2)}$ $VARBL ⟨op 4⟩ $IN
⟨conjunction⟩ → ⟨confrontation⟩ ⇒ I
| ⟨conjunction⟩ ⟨op 3⟩ ⟨confrontation⟩ ⇒ ⟨conjunction⟩ ⟨confrontation⟩ $VARBL ⟨op 3⟩ $IN
⟨disjunction⟩ → ⟨conjunction⟩ ⇒ I
| ⟨disjunction⟩ ⟨op 2⟩ ⟨conjunction⟩ ⇒ ⟨disjunction⟩ ⟨conjunction⟩ $VARBL ⟨op 2⟩ $IN
⟨assignment⟩ → ⟨disjunction⟩ ⇒ I
| ⟨selection⟩ ⟨op 1⟩ ⟨assignment⟩ ⇒ ⟨selection⟩ ⟨assignment⟩ $VARBL ⟨op 1⟩ $IN (32)

The complete table of standard ALGOL 68 operators is given in section 8.4.2 of (11). In rules (32) above, we use operator categories ⟨op 1⟩, . . . , ⟨op 10⟩ to replace the usual arithmetic, logical, and relational operators that appear in similar grammars. For the translator to know which category an operator belongs to, it must have a table of legal operators similar to its nametable, and with each operator will be an associated level number. To each of these operators there corresponds either a standard intermediate-language operation (in which case, the intermediate language operation is written into the translated program) or a procedure definition (in which case the procedure call "$VARBL ⟨op$_n$⟩ $IN" is written into the translated program). Procedures defining the standard operations and their effects when executed are given in section 10.2 of (11).

It should be mentioned that we would include several operations that are not in the ALGOL 68 table. For example, the standard operations of ⟨confrontation⟩ are the relational "=" and "≠". In addition to these standard operators at that level, the ALGOL 68 conformity symbol "::" (which checks whether two expressions are of the same mode) and the ALGOL 68 identity symbol ":=:" (which asks whether two expressions yield references to the same ⟨name⟩) are included because they are used in essentially the same way as "=" and "≠".

Note also that the definition of a jump instruction in ALGOL 68 is put into the ⟨unary⟩ rule of (32) because its precedence in the language is compatible with that level of the grammar. However, the "*go to*" operation is most emphatically *not* redefinable, and so is listed separately.

OPERATOR DEFINITIONS AND DECLARATIONS

Now that we have seen a syntax for expressions, we can discuss the syntax of operator declarations and

priority declarations in ALGOL 68. A priority declaration has the form

⟨priority decl.⟩ → *priority* ⟨operator⟩ = ⟨priority⟩
| ⟨priority decl.⟩, ⟨operator⟩ = ⟨priority⟩ (33)

A priority declaration is not translated, since its role is to provide information to the operator table of the translator. Naturally, the ⟨priority⟩ is some ⟨integer⟩ from 1 to 10:

⟨priority⟩ → 1 | 2 | ··· | 10 (34)

An operator declaration takes the form

⟨operator decl.⟩ → *op* ⟨operator⟩ = (⟨p. list⟩) ⟨indication⟩ *expr* ⟨assignment⟩
⇒ $NEW ⟨operator⟩ $VARBL ⟨operator⟩ .$ ⟨p. list⟩ ⟨assignment⟩ $. = (35)

In (35), the translator enters the new ⟨operator⟩ name onto its operator table and translates the operator declaration as a new procedure definition. Although the ⟨p. list⟩ mechanism is used for simplicity in the translation process, an operator declaration is meaningless unless the ⟨p. list⟩ consists of only one or two parameters. Moreover, as in ALGOL 68, it is assumed that all unary operators have a non-redefinable priority of 10. This is because of the ambiguity that would result if a programmer attempted to redefine the precedence of unary "+" or "−", where the binary addition and subtraction have the same denotations.

As an example of an operator declaration, we give here our version of subtraction with real operands in ALGOL 68:

op- = (*ref real a, b*) *real expr val a* minus *val b*;
op- = (*ref real a, real b*) *real expr val a* minus *b*;
op- = (*real a, ref real b*) *real expr a* minus *val b*;
op- = (*real a, b*) *real expr a* minus *b*;

Here, the operator "−" is translated into the intermediate-language command for subtracting the two topmost values on the run-time operand stack. This definition should be compared with definition 10.2.4(*g*) of the ALGOL 68 report (11), where the definition of subtraction is given in words and then addition and negation are programmed in terms of this subtraction operator.

MODE DECLARATIONS

As a complement to the facility for defining new expression operators, ALGOL 68 allows the definition of new data types and structures of data types. This is accomplished by the mode declaration:

⟨mode decl.⟩ → *mode* ⟨indicant⟩ = ⟨indication⟩
⇒$NEW⟨indicant⟩$VARBL⟨indicant⟩⟨indication⟩= (36)

Here, the translation of ⟨indication⟩ produces some initial value, array, or structure. The ⟨indicant⟩ is translated as though it were a newly declared program variable, and the translation of ⟨indication⟩ becomes its value. This means that, along with its name table and operator table, the translator must have a table of indicants that stores links at translation to any structures that may be assigned to indicants. Thus, when a variable is logically subscripted whose mode is a structure, the translator can find its own prototype copy of that structure and supply appropriate numerical subscripts in the translation.

Because the ⟨indicant⟩ is treated as a program variable by the run-time system (and as a mode declarator by the translator), the translation of ⟨indicant⟩ given in rules (5) presents to the $COPY procedure a list which is copied and assigned to all variables later declared to be of the mode given by the ⟨indicant⟩.

PROGRAM STRUCTURE OF ALGOL 68

At this point, we can complete our description of ALGOL 68 program structure and fill in some remaining details concerning implementation of the language: First, we can draw together the different declarations outlined so far:

⟨declaration⟩ → ⟨mode decl.⟩ ⇒ I
| ⟨operator decl.⟩ ⇒ I
| ⟨priority decl.⟩ ⇒ I
| ⟨type decl.⟩ ⇒ I (37)

We can then complete our definition of a program ⟨statement⟩:

⟨statementsequence⟩ → ⟨labelstat.⟩ ⇒ I
| ⟨statementsequence⟩; ⟨labelstat.⟩ ⇒ I (38)

The effect of the semicolon in rules (38) should not be ignored. Its effect as an intermediate-language command is to unstack the topmost operand of the run-time operand stack. Because of the semicolon, the ⟨block⟩

$$(x := x + 1;\quad 2 + x)$$

causes the intermediate value "$x + 1$" to be unstacked, but leaves the value of "$2 + x$" on the operand stack

as the value of the ⟨block⟩. This same mechanism is used for returning function values, since the last value on the operand stack before exit from a procedure is its value.

Following the scheme of precedence in our syntax, we next define labeled statements:

⟨labelstat⟩ → ⟨statement⟩ ⇒ I
| ⟨name⟩: ⟨labelstat⟩
⇒ $LABEL ⟨name⟩ ↳ ↑ ⟨labelstat⟩ (39)

Here, the sequence "⟨name⟩:" invokes a label declaration in the translated program. The notation "↳↑" means that the value of the translated program location counter plus one is inserted following "$LABEL ⟨name⟩" in the translated program. Strictly speaking, the translator will also have a mechanism for writing out

$LABEL ⟨name⟩

whenever a go to statement is translated before the appropriate label is encountered. In addition, it should be noted that program labels are treated as variable names by the syntax of (15), (16), (19), (30) and (32).

Next, we have a syntax for ⟨statement⟩:

⟨statement⟩ → ⟨assignment⟩
⇒ I

⟨statement⟩ → *for* ⟨selection⟩ *from* ⟨clause⟩(1) *by* ⟨clause⟩(2) *to* ⟨clause⟩(3) *do* ⟨statement⟩
⇒ ⟨selection⟩ ⟨clause⟩(1) ⟨clause⟩(2) ⟨clause⟩(3).$↳⟨statement⟩$.↑ $VARBL $FORSL $IN

⟨statement⟩ → *from* ⟨clause⟩(1) *by* ⟨clause⟩(2) *to* ⟨clause⟩(3) *do* ⟨statement⟩
⇒ ⟨clause⟩(1) ⟨clause⟩(2) ⟨clause⟩(3) .$↳⟨statement⟩$.↑ $VARBL $FOR $IN

⟨statement⟩ → *while* ⟨clause⟩ *do* ⟨statement⟩
⇒ .$↳⟨clause⟩$.↑ ·$↳⟨statement⟩$.↑ $VARBL $WHILE $IN (40)

Listings of the system procedures "$FORSL," "$FOR," and "$WHILE" can be found in Appendix 1.

With these forty sets of rules, we have completed a description of the essential features of ALGOL 68. Missing from the syntax is any built-in procedure for input-output, as well as any description of formatting. As the language is described in this report, formatless input and output procedures for this language can be written that are quite similar to the procedures given in section 10.5.2 of the ALGOL 68 report (11). Such input-output routines have been programmed for the CDC–6500 computer at Purdue University, and are currently being tested together with the remaining components of the ALGOL 68 translator.

BIBLIOGRAPHY

1 R W FLOYD
A descriptive language for symbol manipulation
J ACM No 8 pp 579–584 October 1961

2 E T IRONS
A syntax-directed compiler for ALGOL 60
Communications of the ACM Vol 4 pp 51–55 January 1961

3 P M LEWIS R E STEARNS
Syntax-directed transduction
J ACM No 15 pp 465–488 July 1968

4 D L MILLS
The syntactic structure of MAD/1
Defense Documentation Center Report No Ad-671–683 June 1968

5 P NAUR editor
Revised report on the algorithmic language ALGOL 60
Communications of the ACM No 6 pp 1–17 January 1963

6 V B SCHNEIDER
The design of processors for context-free languages
National Science Foundation Memorandum Department of Industrial Engineering and Management Science Northwestern University Evanston Illinois August 1965

7 V B SCHNEIDER
Pushdown-store processors of context-free languages
Doctoral Dissertation Northwestern University Evanston Illinois 1966

8 V B SCHNEIDER
A system for designing fast programming language translators
Technical Report 68–76 Computer Science Center The University of Maryland College Park Maryland July 1968
Also in Proc Spring Joint Computer Conference 1969

9 V B SCHNEIDER
A translator system for the EULER programming language
Technical Report 69–85 Computer Science Center The University of Maryland College Park Maryland January 1969

10 V B SCHNEIDER
Some syntactic methods for specifying extendible programming languages
Proc Fall Joint Computer Conference 1969

11 VAN WIJNGAARDEN editor
Report on the algorithmic language ALGOL 68
Mathematisch Centrum Report MR-101 Amsterdam February 1969

12 J WEIZENBAUM
A symmetric list processor
Communications of the ACM No 6 p 524 September 1963
13 N WIRTH
A generalization of ALGOL
Communications of the ACM No 6 pp 547–554 September 1963
14 N WIRTH H WEBER
EULER: A generalization of ALGOL and its formal definition: Parts I & II
Communications of the ACM No 3 pp 13–25 and 89–99 January–February 1966

APPENDIX 1—SYSTEM PROCEDURES USED

The following "system procedures," with the exception of the $XEQUT routine, are written in Wirth and Weber EULER.[14] It is understood that their translated versions are supplied by the ALGOL 68 translator to the run-time interpreter system as part of the ⟨standard prelude⟩ of a translated ⟨program⟩. Hence, these procedures are globally defined in every translated ALGOL 68 program.

```
$WHILE ← 'formal clause; formal stat; if clause then
            (stat; $WHILE ('clause', 'stat')) else Ω';

$FORSL ← 'formal var; formal from;
          formal by; formal to; formal stat;
          begin new sign; label cycle;
          sign ← if by <0 then −1 else +1;
          var. ← from;
          cycle: if (to − var ·) × sign >0
          then begin stat; var · ← var · + by;
          go to cycle end else Ω end';

$FOR ← 'formal from; formal by;
       formal to; formal stat;
       begin new index;
        $FORSL (@ index, from, by, to, 'stat') end';

$CASE ← 'formal subscript; formal statementlist;
         statementlist [subscript]';

$COPY ← 'formal structure; formal var;
         var · ← if islist structure then
         begin new dimension; new index;
         dimension ← list (length structure);
         $FORSL (@ index, 1, 1, length structure,
          '$COPY (structure [index], @ dimension [index])');
               dimension end
               else structure';

$ARRAY ← 'formal boundslist; formal value;
          begin new dimension; new index;
          dimension ← list boundslist [1];
          if length (boundslist) = 1
          then $FORSL (@ index, 1, 1, boundslist [1],
          '$COPY (value, @ dimension [index])')
          else $FORSL (@ index, 1, 1, boundslist [1],
          'dimension [index]
             ← $ARRAY (tail boundslist, value)');
          dimension end';
```

If it were legal in EULER to omit the semicolon between statements, thus leaving the values of preceding statements on the operand stack without erasing them, the $XEQUT procedure could be written in EULER as follows:

```
$XEQUT ← 'formal var; formal paramlist;
          (paramlist [1] if length (paramlist) > 1
          then $XEQUT (var, tail paramlist)
          else var ·)';
```

The effect of the procedure above is to place all the parameters of the procedure call onto the rum-time operand stack, and then call the procedure using the "var·" statement. Since the semicolon is missing between "paramlist [1]" and "*if*", the effect of the procedure call is to recursively place the first element of paramlist onto the operand stack, and successively "pop off" the top of the paramlist in each recursive use of paramlist in the call "$XEQUT (var, *tail* paramlist)." In our intermediate language, the $XEQUT procedure can be (correctly) written as follows:

```
$VARBL $XEQUT .$37 $FORMA PARLST
$FORMA VAR $VARBL PARLST $NUMBR 1)
                                    $IN
$VARBL PARLST $IN $LENGT $NUMBR 1
                                    $GT
$IF 13 $VARBL VAR $IN $VARBL PARLST
                                 $IN $TAIL
$VARBL $XEQUT $IN $THEN 5 $VARBL VAR
                                 $IN $IN $ .=;
```

BALM—An extendable list-processing language*

by MALCOLM C. HARRISON

New York University
New York, New York

INTRODUCTION

The LISP 1.5 programming language[1] has emerged as one of the preferred languages for writing complex programs,[2] as well as an important theoretical tool.[3,4] Among other things, the ability of LISP to treat programs as data and vice versa has made it a prime choice as a host for a number of experimental languages.[5,6] However, even the most enthusiastic LISP programmers admit that the language is cumbersome in the extreme.

A couple of attempts[7,8] have been made to permit a more natural form of input language for LISP, but these are not widely available. The most ambitious of these, the LISP 2 project, bogged down in the search for efficiency.

The system described here is a less ambitious attempt to bring list-processing to the masses, as well as to create a seductive and extendable language. The name BALM is actually an acronym (**B**lock **A**nd **L**ist **M**anipulator) but is also intended to imply that its use shoucd produce a soothing effect on the worried programmer. The system has the following features:

1. An Algol-like input language, which is translated into an intermediate language prior to execution.
2. Data-objects of type list, vector and string, with a simple external representation for reading and printing and with appropriate operations.
3. The provision for changing or extending the language by the addition of new prefix or infix operators, together with macros for specifying their translation into the intermediate language.
4. Availability of a batch version and a conversational version with basic file editing facilities.

The intermediate language is actually a form of LISP 1.5 which has been extended by the incorporation of new data-types corresponding to vector, string and entry-point. The interpreter is a somewhat smoother and more general version of the LISP 1.5 interpreter, using value-calls rather than an association-list for looking up bindings, and no distinction between functional and other bindings. The system is implemented in a mixture of Fortran (!) and MLISP, a machine-independent macro-language similar to LISP which is translated by a standard macro-assembler. New routines written in Fortran or MLISP can be added by the user, though if Fortran is used a certain amount of implementation knowledge is necessary.

The description given here is of necessity incomplete because of the flexible nature of the system. In practice it is expected that a number of different dialects will evolve, with different sets of statement forms, operators, and procedures. What is described here is a fairly natural implementation of basic features of the intermediate language which will probably form the basis from which other dialects will grow. We will illustrate the facilities by example rather than by giving a formal description, which can hopefully be obtained from the manual.[9]

OVERVIEW OF BALM FEATURES

A BALM program consists of a sequence of commands separated by semi-colons. Each command will be executed before the next one is read. The user can submit his program either as a deck of cards, or type it in directly from a teletype. When submitted as a card deck, any data required by the command should follow the command immediately, and on the output a listing of the cards will appear, interspersed with any printed output resulting from a command. When a teletype is used, just the output requested will appear.

Variables in BALM do not have a type associated with them, so each variable can be assigned any value. The command:

```
A = 1.2;
```

* This work was done under AEC contract no. AT(30-1)-1480.

would assign the value 1.2 to A, while:

```
PRINT(A);
```

would print out:

```
1.2
```

Arithmetic operations are expressed in the usual way, so:

```
X = 2 * A + 3; PRINT(X);
```

would print:

```
5.4
```

Automatic type conversion is done where necessary.

A " symbol is used to allow the input of *lists*. Thus:

```
A = "(A(B C)D);
PRINT(HD TL A);
```

would print:

```
(B C)
```

The prefix operators HD and TL have the same effect as the functions CAR and CDR in LISP, giving respectively the first element of a *list* and the *list* without its first element. The LISP CONS operator is available either as a procedure, or as an infix colon associating to the right. Thus:

```
A = "A:"(B C):"D:NIL;
```

would also assign the *list* "(A(B C)D) to A.

Vectors can be input in a notation similar to that for *lists*, but using square brackets instead of parentheses. Elements of *vectors* are accessed by indexing. Thus:

```
V = "[A[B C]D]; PRINT(V[2]);
```

would print:

```
[B C]
```

Lists can be members of *vectors*, and vice versa, so:

```
PRINT(TL"(A(B C)D));
```

would print:

```
((B C) D)
```

while:

```
PRINT("[A (B C) D][2]);
```

would print:

```
(B C)
```

A non-rectangular matrix can be expressed as a *vector* of *vectors*:

```
W = "[[1][2 3][4 5 6]];
```

and elements can be extracted by indexing. Thus:

```
PRINT(W[2]);
```

would print:

```
[2 3]
```

Any expression can be indexed so that repeated indexing can be used to extract elements of matrices. Thus:

```
PRINT(W[2][1]);
```

would print out:

```
2
```

Assignments to *vector* elements are straightforward:

```
W[2][1] = "(A B);
```

A whole *vector* or *list* can be assigned from one variable to another variable in a single assignment, of course, but then any operation which changes a component of one will change a component of the other. If this is not desired, the *vector* or *list* should be copied before the assignment:

```
Z = COPY (W);
```

subsequent changes to Z will then not affect W.

An arbitrary structure can be broken up into its constituent parts by the procedure BREAKUP. This takes two arguments, a structure whose elements are constants or variables, and a structure to be broken up. Parts of the second structure corresponding to variables in the first structure are assigned as the values of those variables, while constants must match. If the structures cannot be matched, the BREAKUP procedure is terminated and gives the value NIL. Otherwise it has the value TRUE. For example:

```
BREAKUP ("(A B), "((C C) (D D)));
```

will give the value TRUE and will assign "(C C) to A and "(D D) to B. Either structure can involve vectors, and constants in the first structure are specified by preceding them with the quote mark ("). Thus:

```
BREAKUP ("[A "B C], "[[X X] B [Y Y]]);
```

will have the value TRUE and will assign "[X X] to A and "[Y Y] to B. The converse of BREAKUP is CONSTRUCT, which is given a single structure whose elements are variables, and which will construct the same structure but with variables replaced by their values. Thus:

```
X = "(A B); Y = [C D];
PRINT(CONSTRUCT ("(X Y)));
```

will print ((A B) [C D]). These two procedures allow convenient forms such as:

```
IF BREAKUP ("(A "+ B), X) THEN RETURN
(CONSTRUCT("[A B "PLUS]));
```

BREAKUP and CONSTRUCT are quite efficient,

and should be used in preference to the more primitive operations whenever possible.

Character strings of arbitrary length can be specified:

```
C = ⟨EXAMPLE OF A STRING⟩;
```

They can be concatenated, or have substrings extracted. Thus:

```
D = C → ⟨AND ANOTHER⟩;
E = SUBSTR(D,9,4); PRINT(E);
```

would print

```
⟨OF A⟩
```

The BALM system allows the user to assign *properties* to variables. A *property* consists of a name and a value. For example, the command:

```
"VAR PROP "ABCD = ⟨STR⟩;
```

assigns the *property* called ABCD with an associated value of ⟨STR⟩ to the variable VAR. Similarly:

```
X = "VAR PROP "ABCD;
```

will set the value of X to the value of the *property* ABCD of variable VAR. A variable can have any number of *properties* and any number of variables can have the same *property*.

There is complete garbage collection of all inaccessible objects in the system, so the user does not need to keep track of particular *lists* or *vectors*. Procedures are available for creating *lists* or *vectors* with values of expressions as their elements, with storage being allocated dynamically:

```
LL = LIST(Z + Q, ABC, S⟨XY⟩);
VV = VECTOR(X + W, ABC, S⟨XY⟩);
```

A *procedure* in BALM is simply another kind of data-object which can be assigned as the value of a variable. The variable can then be used to invoke the *procedure* in the usual way. The statement:

```
SUMSQ = PROC(X, Y), X ↑ 2 + Y ↑ 2 END;
```

assigns a *procedure* which returns as its value the sum of the squares of its two arguments. The translator translates the PROC . . END part into the appropriate internal form, which is assigned to SUMSQ. In fact this is simply a *list*, which could equally well have been calculated as the value of an expression. The *procedure* can subsequently be applied in the usual way. For example:

```
PRINT(5 + SUMSQ(2, 3) + 0.5);
```

would print:

```
18.5
```

Instead of assigning a *procedure* as the value of a variable, we can simply apply it, so that:

```
X = 5 + PROC(X, Y), X ↑ 2 + Y ↑ 2 END(2, 3) + 0.5;
```

would assign 5 + 13 + 0.5 = 18.5 as the value of X. Note that a *procedure* can accept any data-object as an argument, and can produce any data-object as its result, including *vectors*, *lists*, *strings* and *procedures*. Thus it is possible to write:

```
M = MSUM(M1, MPROD(M2, M3) );
```

where M1, M2, M3, and M are matrices. *Procedures* can be recursive, of course.

Analogous to *procedures* we can also compute with *expressions*. The statement

```
E = EXPR A + B END;
```

would assign the *expression* A + B, not its value, to E. Subsequently, values could be assigned to A and B, and E evaluated:

```
A = 1; B = 2.2; V = EVAL(E);
```

EVAL(E) could also have been written as $E.

A *procedure* is simply an expression with certain variables specified as arguments. The most useful expression for *procedure* definitions is the *block*, which is similar to that used in ALGOL, but can have a value. The statement:

```
REVERSE = PROC(L),
          BEGIN(X),
          COMMENT ⟨FIRST TEST FOR
            ATOMIC ARGUMENT⟩
          IF ATOM(L) THEN RETURN(L),
          COMMENT ⟨OTHERWISE ENTER
            REVERSING LOOP⟩
          X = NIL,
          COMMENT ⟨EACH TIME ROUND
            REMOVE ELEMENT FROM L,
            REVERSE IT, AND PUT AT
            BEGINNING OF X⟩
NXT,      IF NULL(L) THEN RETURN(X),
          X = REVERSE(HD L):X,
          L = TL L, GO NXT
          END END;
```

shows the use of a *block* delimited by BEGIN and END in defining a *procedure* REVERSE which reverses a *list* at all levels. The COMMENT operator can follow any infix operator, and will cause the following data-item to be ignored.

As well as IF . . . THEN . . . statement there is an IF . . . THEN . . . ELSE . . . as well as an IF . . . THEN . . . ELSEIF . . . THEN . . . etc. Looping statements include a FOR . . . REPEAT . . . as well as a WHILE . . .

REPEAT . . . . A label should be regarded just as a local variable whose value is the internal representation of the statements following it. Accordingly, assignments to labels, and transfers to variables or expressions are legal, and can give the effect of a switch. A *compound* statement without local variables or transfers can be written DO . ., . ., . . END. Of course any of these statements can be used as an expression, giving the appropriate value. Note that a comma is used to separate statements and labels within a *block* and a *compound* statement. The semicolon is interpreted as an end-of command by the system (unless it occurs within a string), even if it occurs within parentheses or brackets. Any unpaired parentheses or brackets will be paired automatically, with a warning message being issued.

EXTENDABILITY

The TRANSLATE procedure used by BALM to translate statements into the appropriate internal form is particularly simple, consisting of a precedence analysis pass followed by a macro-expansion pass. Built-in syntax is provided only for parenthesized subexpressions, comments, the quote operator, the NOOP operator, procedure calls, and indexing. All other syntax information is provided in the form of three lists which are the values of the variables UNARYLIST, INFIXLIST, and MACROLIST. The user can manipulate these lists as he wishes, by adding, deleting, or changing operators or macros.

Operators are categorized as *unary*, *bracket*, or *infix*, and have precedence values, and a procedure (or macro) associated with them. Examples of *unary* operators are −, HD and IF, while *infix* operators include +, THEN, and =. *Bracket* operators are similar to *unary* operators but require a terminating *infix* operator which is ignored. Examples of *bracket* operators are BEGIN and PROC, which both can be terminated by the *infix* operator END.

New operators can be defined by the *procedures* UNARY, BRACKET, or INFIX. These add appropriate entries onto UNARYLIST or INFIXLIST. For example the statement:

```
UNARY("PR, 150, "PRINT);
```

would establish the *unary* operator PR with priority 150 as being the same as the *procedure* PRINT. Thus we could subsequently write PR A instead of PRINT(A). Similarly we could define an *infix* operator by

```
INFIX("→, 49, 50, "APPEND);
```

to allow an infix append operation. The numbers 49 and 50 are the precedences of the operator when it is considered as a left-hand and right-hand operator respectively, so that an expression such as A → B → C will be analyzed as though it were A → (B → C)

The output of the precedence analysis is a tree expressed as a *list* in which the first element of each *list* or sublist is an operator or macro. For example, the statement:

```
SQ = PROC(X), X * X END;
```

would be input as the *list*:

```
(SQ = PROC (X) , X * X END)
```

and would be analyzed into:

```
(SETQ SQ (PROC (COMMA X (TIMES X X))))
```

This would then be expanded by the macro-expander, giving:

```
(SETQ SQ (QUOTE (LAMBDA (X) (TIMES X X))))
```

the appropriate internal form. This would then be evaluated, having the same effect as the statement:

```
SQ = "(LAMBDA(X) (TIMES X X));
```

which would in fact be translated into the same thing.

The macro-expander is a function EXPAND which is given the syntax tree as its argument. It is actually defined as:

```
EXPAND = PROC(TR),
         BEGIN(Y),
         IF ATOM(TR) THEN RETURN(TR),
         Y = LOOKUP(HD TR, MACROLIST),
         IF NULL(Y) THEN RETURN
           (MAPCAR(TR, EXPAND)),
         RETURN(Y(TR))
         END END;
```

That is, if the top-level operator is a macro, it is applied to the whole tree. Otherwise EXPAND is applied to each of the subtrees recursively. Most operators will not require macros because the output of the precedence analysis is in the correct form. However, operators such as IF, THEN, FOR, PROC . . . etc. require their arguments to be put in the correct form for the interpreter. For instance, the IF macro can be defined:

```
MIF = PROC(TR),
      BEGIN(X),
      X = HD TL TR,
      IF HD X ≡ "THEN THEN RETURN
        ("COND: LIST(EXPAND(HD TL X),
        EXPAND(HD TL TL X)):NIL),
      RETURN("COND:EXPAND(X))
      END END;
```

where recursive calls to EXPAND are used to transform subtrees in the appropriate way. The statement:

MACRO("IF, MIF);

would associate the macro MIF with the operator IF.

We can think of this expansion procedure as top-down, in the sense that a higher level macro in the tree is expanded before a lower level macro. In fact the higher level macro can process the tree in any way. This may include not processing the tree at all (as is done by the QUOTE macro), or expanding selected subtrees in a standard or non-standard way. A macro can even act as a translator of a special-purpose sub-language which is quite different from BALM. For example, the expression:

SNOBOL "(X ⟨ABC⟩ ARB Y PP(I) = Y :F(FAIL))

is perfectly legitimate in BALM, and could be translated into the appropriate internal form by a macro associated with the prefix operator SNOBOL.

One particular outcome of this expansion procedure is the ability to write other than simple variables on the left-hand-side of assignment statements. These are conveniently handled by a macro associated with the assignment operator which checks the expression on the left-hand-side and modifies the syntax tree accordingly. It is this mechanism which permits an element of a vector to appear on the left-hand-side, and also such statements as:

HD X = Y;

which will be translated as though it had been written:

RPLACA(X, Y);

The assignment macro currently in use looks up the top level operator found on the left-hand-side in a *list* LMACROLIST, applying any macro associated with the operator to the tree representing the assignment statement. The set of expressions which can be handled on the left-hand-side can easily be extended by adding entries to LMACROLIST. For example:

LMACRO("PROC,MPROC);

could be used to add the left-hand-side macro MPROC to permit assignments such as:

PROC PPP(X,Y) = EXPR . . . END;

as an alternative way of defining a *procedure.*

Note that the essential properties of the system are those of the intermediate language, the most important of which is its ability to treat data as program, and thus to preprocess its program. Even the TRANSLATE procedure described above can be ignored and the user's own translator substituted. Of course this will require a different level of expertise on the part of the programmer than simply the addition of new operators. However, the translator, which takes about 2000 words on the CDC 6600, is only about 250 cards, and quite straightforward, so this is not an unlikely possibility.

In summary, the BALM system permits extendability in a number of different ways:

1) By addition of user-defined *procedures.*
2) By the definition of unary or infix operators.
3) By the definition of macros.
4) By the use of a user-defined translator.

Procedures, macros and translators can be written with the full power of BALM, or in MLISP or assembly language.

ACKNOWLEDGMENTS

Much of the coding of the current version of BALM has been done by Douglas Albert and Jeffrey Rubin, both of whom have made substantial contributions to its development.

REFERENCES

1 J McCARTHY et al
Lisp 1.5 programmers manual
MIT Press 1962
2 M MINSKY
Semantic information processing
MIT Press 1968
3 J PAINTER
Semantic correctness of a compiler for an Algol-like language
A I Memo 44 Stanford University 1967
4 P LANDIN
The mechanical evaluation of expressions
Computer Journal January 1964
5 D BOBROW J WEIZENBAUM
List processing and extension of language facility by embedding
IEEE Trans on Elec Comp EC-13 Aug 1964
6 C ENGLEMAN
Mathlab—A program for on-line machine assistance in symbolic conputations
Proc FJCC 1965
7 L P DEUTCH
940 LISP reference manual
University of California Berkeley February 1966
8 P ABRAHAMS et al
The Lisp 2 programming language and system
Proc FJCC 1966
9 M HARRISON
BALM users manual
Courant Inst Math Sci New York University

Design and organization of a translator for a partial differential equation language

by ALFONSO F. CARDENAS and WALTER J. KARPLUS

University of California
Los Angeles, California

INTRODUCTION

In recent years a variety of techniques for the design and implementation of translators for problem-oriented programming languages has been developed. A number of these employ a high-level programming language such as FORTRAN as the target language, rather than translating directly into assembly language or machine code. The purpose of the present paper is to demonstrate the unique advantages which can be realized by making PL/1 the target language and by utilizing the so-called preprocessor (or compile-time) facilities of PL/1. This approach has been successfully used in the design of a translator for PDEL, a special-purpose language for the solution of the partial differential equations of classical physics. Details regarding the language and its application have been presented in an earlier paper[1] and are, therefore, only briefly summarized in the next section. The overall structure and philosophy of the translator are described in the third section, while more detailed aspects of syntax analysis and code generation are described in the fourth section. In the final section a quantitative evaluation of the translator is presented.

REVIEW OF THE BASIC FEATURES OF PDEL

PDEL was designed at the University of California at Los Angeles and implemented in its basic form in 1968. Its purpose is to facilitate the solution of those partial differential equations which are of particular importance to engineers. These include particularly the elliptic equations which characterize potential fields, the parabolic equations which characterize heat transfer and diffusion, the hyperbolic equations which characterize wave phenomena, and the biharmonic equations which arise in elasticity problems. The classes of problems which could be handled by the original translator are shown in Table I. Subsequent extensions of the translator have been directed toward permitting the treatment of fields in three space-dimensions, the inclusion of a wider variety of boundary conditions, and the handling of singularities, moving boundaries, and other special features.

The numerical treatment of such partial differential equations most often proceeds from finite difference approximations. The time-space continuum is replaced by an array of regularly-spaced points in one, two, three, or four dimensions, and sets of algebraic equations are solved simultaneously to provide solutions. In the case of elliptic equations, the solutions for all points are obtained simultaneously; in the case of transient field problems, solutions are obtained sequentially for successive steps in the time domain. A variety of algorithms are available for the solution of the difference equations. In order to obviate the problem of computational stability, the catastrophic accumulation of round-off errors, so-called implicit methods are usually preferred. Even so, the numerical analyst must make a judicious choice from among several practical algorithms, a choice which sometimes has a decisive effect upon computer execution time. This choice depends, of course, upon the specific type of equation under study, but is also influenced by the geometry (whether the field has regular boundaries), the parameter distribution (whether the field is linear or nonlinear, or constant or variable parameter), the required solution accuracy, and by the size of available fast memory.

A language designed to solve partial differential equations must, therefore, provide various algorithms for the different types of equations; these algorithms may necessarily involve the construction of lengthy subroutines. To avoid waste of computer time, only the subroutine corresponding to the problem at hand should be produced and compiled. The preprocessor

facilities of PL/1 are exceptionally well-suited to this end.[2,3] Accordingly, PL/1 was selected as the target language of the translator, the translator was written in preprocessor PL/1, and all legal PDEL statements were designed so as to be legal PL/1 preprocessor statements.

A typical PDEL program may contain the following statements, expressed in a syntax chosen to be readily comprehensible to engineers:

1. Definition of the equation to be solved, i.e., the mathematical form of the equation including parameter identifiers, the order of the partial derivatives, Poissonian terms, etc.;
2. Parameter specification, which may be constants or functions of the dependent and independent problem variables;
3. Specifications of the finite difference grid spacing for the space and the time variables;
4. Description of the geometry of the field, that is the coordinates in the space domain of the field boundaries;
5. Boundary conditions, which may be of the Dirichlet, Neumann or Fourier types;
6. Selection of one of available algorithms to be employed;
7. Bounds on the number of iterations or the iteration error for iterative algorithms
8. Description of the type and nature of the printout desired.

Default conditions are provided for most of these statements, so that the programmer may choose to omit certain specifications in the program; in this case the translator will make the selection for him. An example of a PDEL program for a simple partial differential equation is presented in Appendix 1, together with a typical print out. To date, over a hundred partial differential equations have been programmed and solved using PDEL.

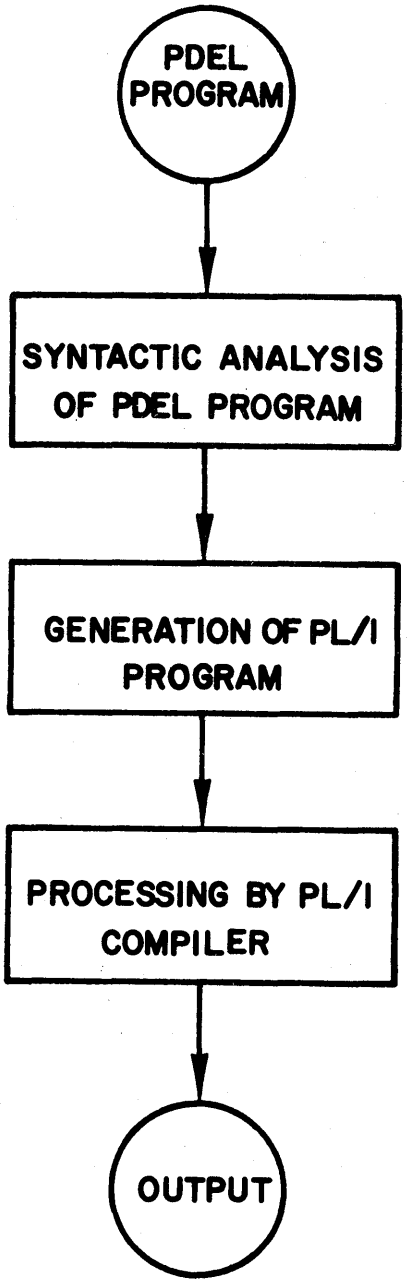


Figure 1—Processing of a PDEL program

GENERAL TRANSLATION APPROACH

As indicated in the preceding section, a translator for a partial differential equations language must contain a selection of algorithms, each suitable for a different class of partial differential equations, different geometries, etc. A study of digital computer programs corresponding to a number of these algorithms indicated that for a given class of equations, approximately 70% to 95% of the total number of program statements was common to all programs. The other 5% to 30% of the statements dealt with the description of parameters, initial and boundary conditions, and geometries which vary from problem to problem. The greater the complexity of the geometry of the field, the greater number of special statements required for characterizing the particular boundary configuration. A key to the successful and economic translation of partial differential equations is the avoidance of the generation of unnecessary code. That is, it is important to assure that only those portions of the translator required for the specific problem to be solved is selected for compilation, and then combined with user-generated statements cor-

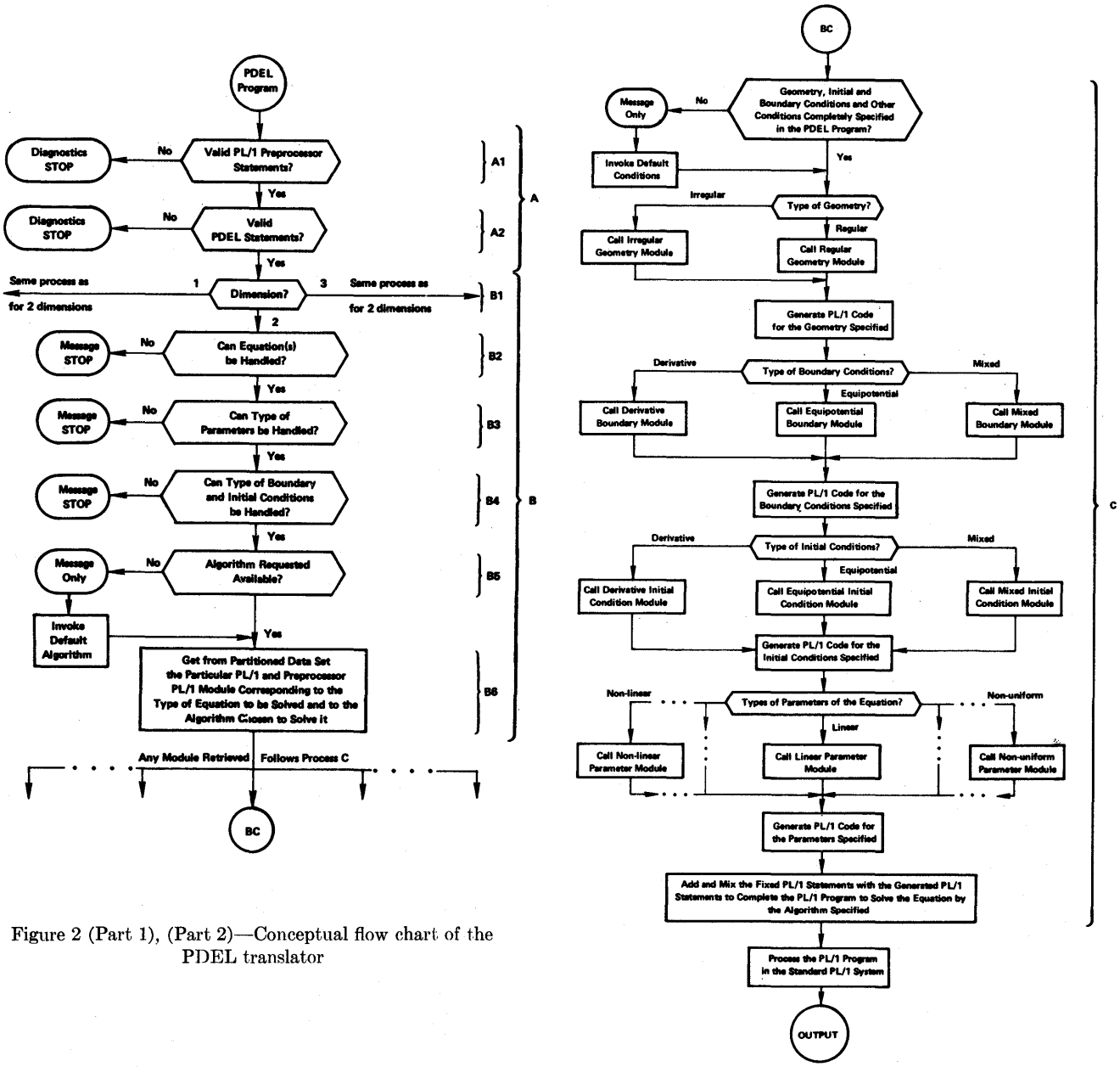


Figure 2 (Part 1), (Part 2)—Conceptual flow chart of the PDEL translator

responding to the 5% to 30% of the program which is specific to the problem being solved.

Among the major problem-oriented languages currently in wide use, only PL/1, by virtue of its compile-time facilities, permits full control over which portions of a program are to be compiled. The PL/1 preprocessor language is a subset of PL/1, with many significant features particularly suited for character-string manipulation and generation, and hence for language translation.[2,3] As the name indicates, the preprocessing phase immediately precedes the actual PL/1 compilation. During this phase, the program is scanned for all statements which contain the % identifier; and only those statements are executed.

The PDEL translator is in effect a preprocessor program which automatically chooses groups or modules of fixed PL/1 statements corresponding to the equation to be solved and the algorithms to be employed. These modules of fixed PL/1 code are stored in secondary storage devices (e.g., disc, data cells), and constitute the 70% to 95% of the statements common to all programs of a given class. The other 5% to 30% of the code is generated during the preprocessing phase by specially designed code generation routines which are a

part of the PDEL translator. If desired, the programmer may also include PL/1 statements (without the % identifier) in his PDEL program. These statements are unaffected by the preprocessing and proceed directly to the PL/1 compiler.

The overall translation process proceeds as shown in Figure 1. The preprocessor phase, which results in the generation of a PL/1 program, is performed in two stages: Syntactic analysis and code generation. This permits the diagnosis of illegal statements, programming errors, and calls for modules not presently available, prior to generation of PL/1 code. The syntactic analysis in turn proceeds in two steps:

1. The standard PL/1 preprocessor analyzes the PDEL program to determine that it is a valid PL/1 preprocessor program.

2. The syntactic analyzers of the PDEL translator analyze the strings on the right hand side of each PDEL statement, so as to detect violations of PDEL syntax.

Figure 2, is a generalized conceptual flow-chart of the PDEL translator. Syntactic analysis is represented by blocks A1 and A2. Blocks B determine whether the required PL/1 modules are available in secondary storage, while block C contains all the PL/1 code generators.

The approximate size of the PDEL program is illustrated in Figure 3, which also indicates the sequence of operations. The two INCLUDE statements appear in the original PDEL program and serve to call the PDEL translator.

A finer view of the operation of the translator is given in the next section.

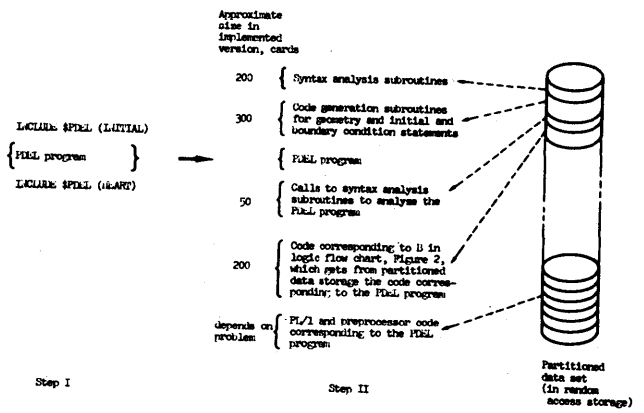


Figure 3—Physical movement of code to translate a PDEL program

DETAILED OPERATION OF THE TRANSLATOR

The whole translator is a set of modules stored in external memory. Each module is a subroutine which may be retrieved from external storage if needed to process the PDEL program. The task of the modules retrieved and the order in which they act is shown in Figure 2. The order in which they are physically retrieved from external storage is shown in Figure 3.

The main modules available are:

(1) Three subroutines which perform the syntactic analysis of PDEL statements, stage A2 in Fig. 2. One of these is the master analysis routine which determines whether or not each PDEL statement is constructed according to the rules of syntax stored in the other two subroutines.

(2) A monitor module which performs all of step B in Fig. 2, that is, it identifies the problem defined in the PDEL program and retrieves from external storage the module made up of the fixed PL/1 statements and PL/1 preprocessor statements corresponding to the problem.

(3) A subroutine which when invoked examines the specification of the geometry of the field in the PDEL program and generates the appropriate set of PL/1 statements used to form the PL/1 program.

(4) A subroutine which examines the boundary conditions specified in the PDEL program and generates the appropriate set of PL/1 statements used to form the PL/1 program.

(5) A subroutine which examines the initial conditions specified in the PDEL program and generates the appropriate set of PL/1 statements used to form the PL/1 program.

(6) A set of subroutines made up of fixed PL/1 code and preprocessor code. Each subroutine corresponds to a given type of equation, and, when processed by the preprocessor, produces the PL/1 program to solve the equation for the particular parameters, initial and boundary conditions, geometry of the field and by the numerical algorithm indicated in the PDEL program.

(7) A set of PL/1 subroutines which produce plots of the solution of the equation, e.g., contour plots.

As shown in Fig. 3, the first of the two INCLUDE statements in a PDEL program retrieves modules (1), (2), (3), (4), and (5). The second INCLUDE statement retrieves the statements that call on modules (1) to do the syntax analysis. If no errors are detected, the monitor module (2) is called and it decides which module (6) to retrieve in order to produce the PL/1 program.

Some details on the syntax analysis and on the manner in which a PL/1 program is typically produced follow.

Syntax analysis

PDEL has been designed such that a valid PDEL program is a valid preprocessor program. Every PDEL statement is a valid PL/1 preprocessor statement, and its general form is (in Backus Naur form):

% ⟨variable name or key word⟩ = '⟨character string whose syntax and meaning is the concern of the PDEL translator⟩';

The PL/1 preprocessor makes sure of this without attaching any meaning to the character string on the right hand side of each PDEL statement. The syntax and interpretation of such strings is the concern of the PDEL translator. The syntax analysis capability of the preprocessor is thus utilized as much as possible. If a different overall translation approach had been taken (e.g., not writing the translator in preprocessor PL/1) or if PDEL were not within the syntax of preprocessor PL/1, then syntax analyzers for the whole PDEL language, rather than for only a part of it, would have had to be designed. Simple precedence syntax analysis techniques, first introduced by Wirth and Weber,[5] are used in the PDEL translator.

Production of a PL/1 program

The manner in which a PL/1 program is typically formed will now be illustrated by examining the production of the PL/1 program corresponding to the PDEL in Appendix 1 to solve the indicated two-dimensional elliptic equation. As already pointed out, the general PDEL translation proceeds conceptually in the logic manner shown in Figure 2 and with the physical movement of code illustrated in Figure 3. At stage B6 in Figure 2 the module for solving the two-dimensional elliptic equation by the specified algorithm (in this case by successive overrelaxation) is retrieved from external storage and sent to the PL/1 preprocessor. This module has the structure indicated in Figure 4, and is typical of all modules to solve partial differential equations. The fixed statements are part of the PL/1 program formed to solve all two-dimensional elliptic equations.

The module is examined by the preprocessor. The preprocessor sends directly to the PL/1 compiler any PL/1 statements that it finds and executes the preprocessor statements. With this in mind and referring to

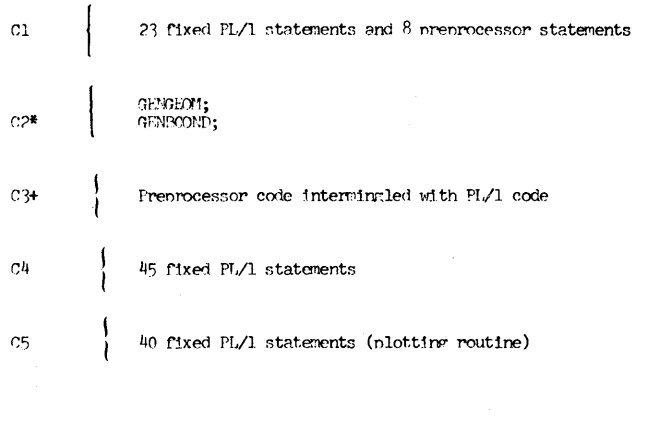


Figure 4—Structure of the module for a two-dimensional elliptic equation

Figure 4, the PL/1 program is formed as follows:

C1: These are mostly PL/1 statements which are sent directly to the PL/1 compiler. They include declaration statements, statements to print out the PDEL program and statements which set up default conditions in case that the geometry and the boundary conditions of the field are not fully specified in the PDEL program. However, a few preprocessor statements are intermingled with this PL/1 code so that when they are executed by the preprocessor they will modify the PL/1 code slightly and thus send the appropriate statements to the PL/1 compiler.

C2: These are preprocessor calls to the preprocessor procedures GEMGEOM and GENBCOND, which, as shown in Step II of Figure 3, have been already retrieved and placed ahead of the PDEL program. GENGEOM is executed and results in a number of PL/1 statements corresponding to the geometry of the field specified in the geometry statement of the PDEL program; these statements control the scan of the numerical algorithm to solve the equation. GENBCOND is also executed and results in PL/1 statements which set up the boundary conditions specified in the boundary condition statement of the PDEL program.

C3: This is a mixture of PL/1 and preprocessor code. When the preprocessor statements are executed they modify the PL/1 code. The result is the heart

TABLE I—Problems that may be solved by the PDEL translator as of June 1969.

TYPE	PARTIAL DIFFERENTIAL EQUATION*	PARAMETERS	INITIAL CONDITIONS	BOUNDARY CONDITIONS
Elliptic 1 dimension	$\frac{\partial}{\partial x}\left(\sigma\frac{\partial\phi}{\partial x}\right)=K$	σ = constant K = $f(\phi)$ $f(x)$	———	$\phi\|_{boundary}$ = constant $\frac{\partial\phi}{\partial n}\|_{boundary}$ = $f(x,\phi)$
Elliptic 2 dimensions	$\frac{\partial}{\partial x}\left(\sigma_x\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\sigma_y\frac{\partial\phi}{\partial y}\right)=K$	σ_x, σ_y, K = constant, $f(x,y)$ K = $f(\phi)$	———	$\phi\|_{boundary}$ = constant, $f(x,y)$ $\frac{\partial\phi}{\partial n}\|_{boundary}$ = constant, $f(x,y,\phi)$
Parabolic 1 dimension	$\frac{\partial}{\partial x}\left(\sigma\frac{\partial\phi}{\partial x}\right)=K_1\frac{\partial\phi}{\partial t}+K_2$	σ, K_1, K_2 = constant, $f(\phi)$	$\phi\|_{t=0}$ = constant, $f(x)$	$\phi\|_{boundary}$ = constant $\frac{\partial\phi}{\partial n}\|_{boundary}$ = $f(x,\phi)$, constant
Parabolic 2 dimensions	$\frac{\partial}{\partial x}\left(\sigma_x\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\sigma_y\frac{\partial\phi}{\partial y}\right)=K\frac{\partial\phi}{\partial t}$	σ_x, σ_y = constant K = $f(\phi)$	$\phi\|_{t=0}$ = constant, $f(x,y)$	$\phi\|_{boundary}$ = constant, $f(x,y)$
Hyperbolic 1 dimension	$\frac{\partial}{\partial x}\left(\sigma\frac{\partial\phi}{\partial x}\right)=\frac{\partial}{\partial t}\left(K_1\frac{\partial\phi}{\partial t}\right)+K_2\frac{\partial\phi}{\partial t}+K_3$	σ, K_3 = constant, $f(x,t)$, $f(\phi)$ K_2, K_1 = constant, $f(x)$, $f(\phi)$	$\phi\|_{t=0}$ = constant, $f(x)$ $\frac{\partial\phi}{\partial t}\|_{t=0}$ = constant, $f(x)$	$\phi\|_{boundary}$ = constant

* The geometry of two dimensional fields can be of any form; the only restriction is that no more than four boundaries may exist when looking in the x direction from any point in the y axis

of the PL/1 program: the PL/1 code directly related to the numerical algorithm used to solve the partial differential equation. These modifications depend on the types of parameters of the problem. The overrelaxation factor and maximum number of iterations specified in the PDEL program are also taken into account here. Actually, a finer view shows that there is a series of nine groupings of mixed PL/1 and preprocessor code. Each grouping contains from fifteen to twenty-five statements and applies to a particular type of parameter of the partial differential equation. The appropriate grouping is chosen automatically and is sent to the preprocessor to produce the necessary PL/1 statements. Figure 5 shows the mixed code corresponding to the example in Appendix 1, and Figure 6 shows the PL/1 code that is then produced. This illustrates in particular the replacement of preprocessor variable names. The preprocessor scans the input in sequence. When a preprocessor statement (identified by a leading % sign) is encountered, it is executed. When a PL/1 statement is scanned, every character string in it that is a preprocessor variable and to which a

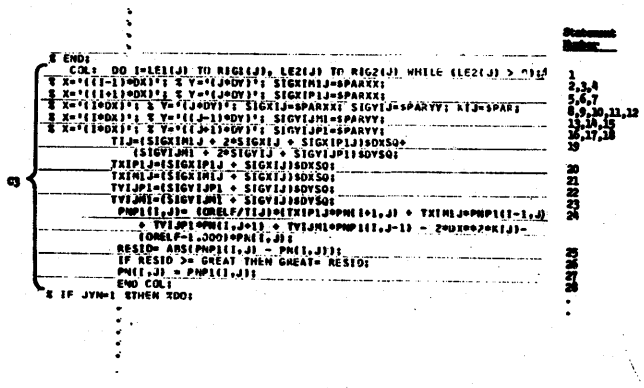


Figure 5—Mixed PL/1 and preprocessor code sent to the PL/1 preprocessor. This code corresponds to the two-dimensional elliptic problem in Appendix 1.

TABLE II—Computer time and size of programs for several arbitrarily selected problems.

EQUATION	PROCESSING TIME*, seconds, on IBM 360/75 with standard PL/1 system				NUMBER OF STATEMENTS	
	Translation** of PDEL	Compilation	Execution	Total	PDEL	PL/1
$\frac{\partial^2\phi}{\partial x^2} = e^{\phi}$	18	26	11	58	11	108
$\frac{\partial}{\partial x}\left((10+y)\frac{\partial\phi}{\partial x}\right) + \frac{\partial}{\partial y}\left((10+y)\frac{\partial\phi}{\partial y}\right) = 0$	39	34	19	93	19	154
$\frac{\partial^2\phi}{\partial x^2} = (0.08/\phi)\frac{\partial\phi}{\partial t}$	12	28	10	51	16	118
$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 4\phi^{-3/4}\frac{\partial\phi}{\partial t}$	23	42	22	91	20	256

* Translation (includes associated overhead) + Compilation (includes associated overhead) + Execution (includes associated overhead) = Total + distributed overhead.

**Translation time was obtained by subtracting the total time required to process separately the PL/1 program generated from the time required to completely process the corresponding PDEL program.

† With arbitrarily selected geometries, initial conditions and boundary conditions of various complexities. The more the complexity the greater the number of PL/1 statements generated.

value (a number or another character string) has been given is replaced by the value. The statement is then sent to the PL/1 compiler. The primed statement numbers in Figure 6 identify statements which are the result of such a replacement activity.

```
                                                                    Statement
                                                                    Number
      ROW: DO J= 1 TO (JMAX-1);
      COL: DO I=LE1(J) TO RIG1(J), LE2(J) TO RIG2(J) WHILE (LF2(J) > 0); 1
           SIGXIMlJ=  10+ (J*DY)   ;                                    8'
           SIGXIPlJ=  10+ (J*DY)   ;                                    9'
           SIGXIJ=  10+ (J*DY)   ; SIGYIJ=  10+ (J*DY)   ; PIJ=  0.0   ;  10',11',12'
           SIGYIJM1=  10+ ((J-1)*DY)   ;                                14'
           SIGYIJP1=  10+ ((J+1)*DY)   ;                                15'
                TIJ=(SIGXIMlJ + 2*SIGXIJ + SIGXIPlJ)  +                 19'
                  (SIGYIJM1 + 2*SIGYIJ + SIGYIJP1)  ;
                TXIP1J=(SIGXIPlJ + SIGXIJ)  ;                           20'
                TXIM1J=(SIGXIMlJ + SIGXIJ)  ;                           21'
G3              TYIJP1=(SIGYIJP1 + SIGYIJ)  ;                           22'
                TYIJM1=(SIGYIJM1 + SIGYIJ)  ;                           23'
                 PNP1(I,J)= (OMELF/TIJ)*(TXIP1J*PN(I+1,J) + TXIM1J*PNP1(I-1,J)  24
                   + TYIJP1*PN(I,J+1) + TYIJ41*PNP1(I,J-1) - 2*DX**2*RIJ)-
                   (OMELF-1.000)*PN(I,J);
                 RESID= ABS(PNP1(I,J) - PN(I,J));                       25
                 IF RESID >= GREAT THEN GREAT= RESID;                   26
                 PN(I,J) = PNP1(I,J);                                   27
                 END COL;                                               28
           END ROW;                                                     .
           IT= IT + 1;                                                  .
           IF IT= ITMAX & GREAT> EPS THEN DO;                           .
                PUT EDIT ('SOLUTION DOES NOT CONVERGE IN ', ITMAX,
                  ' ITERATIONS') (A,F(5), A) SKIP(5);
                GO TO E1;
                END;
           IF GREAT > EPS THEN GO TO E1;
                ELSE DO;
                     PUT EDIT ('THE SOLUTION CONVERGES WITHIN ',
                       EPS,' IN ',IT,' ITERATIONS, AND, FOR EACH
      GRID POINT INDICATED, IT IS:'
      )(A,E(12,5),A,E(12,5),A) SKIP(6);
```

Figure 6—PL/1 code sent to the PL/1 compiler. This code corresponds to the two-dimensional elliptic problem in Appendix 1.

C4: These statements are sent directly to the PL/1 compiler. They include: (a) output statements to print out the tabular solution of the equation in the manner specified in the PDEL output statements, and (b) other statements which complete the PL/1 program.

C5: If a plot of the solution is ordered in the PDEL program, the module containing the appropriate PL/1 subroutine to plot two-dimensional solutions is retrieved and sent to the PL/1 compiler.

Thus the complete PL/1 program is produced. It is then compiled and executed like any regular PL/1 program to produce the solution to the equation. The number of PL/1 statements involved in this example is indicated in Figure 4.

EVALUATION OF THE PDEL TRANSLATOR

As yet no generally accepted criteria for the evaluation and comparison of translators are available in the computer software area. Qualitative discussions gen-

erally center on: compatibility, diagnostic capability, and efficiency. Some comments as to the characteristics of the PDEL translator as far as these three qualities are concerned, appears in order.

Compatibility with other computers

The PDEL translator is compatible with any computer for which there exists a standard PL/1 compiler and sufficient external random access memory. The translator itself is written in preprocessor PL/1, a high-level language, and makes no reference to unique hardware features.

Diagnostic capability

The preprocessor PL/1 syntax analyzer (furnished with the PL/1 compiler) and the PDEL syntax analyzer function effectively to detect any programming errors involving the violation of syntactic rules. Errors which can be detected only at execution time (for example, overflow and requests for excessively large arrays) create difficulties, difficulties which arise in the use of most higher-level programming languages.

Efficiency

A working PDEL translator capable of solving the problems indicated in Table 1, has been in operation since 1968. It contains approximately 3,000 cards. The preprocessing, compilation, and execution time for four arbitrarily selected field problems are summarized in Table II together with the number of PDEL and PL/1 statements required in each case. Improvements in some of these figures have been effected recently by evolutionary changes in the translator.

REFERENCES

1 A F CARDENAS W J KARPLUS
PDEL—A language for partial differential equations
Communications of the ACM 1970

2 *PL/1 language specifications*
IBM Systems Reference Library Form C28-8201 1968

3 R L GAUTHIER
PL/1 compile time facilities
Datamation pp 32–34 December 1968

4 J A FELDMAN D GRIES
Translator writing systems
Communications of the ACM pp 77–113 February 1968

5 N WIRTH H WEBER
EULER; A generation of ALGOL and its formal definition—Part I and Part II
Communications of the ACM pp 13–25 and 89–99 February 1966

6 J A LEE
The anatomy of a compiler
Reinhold Publishing Corporation 1967

7 F R HOPGOOD
Compiling techniques
American Elsevier Publishing Company 1969

8 L PRESSER
Translation of programming languages
Survey of Computer Science edited by A F Cardenas M A Marin and L Presser
To be published

9 J CEA B NIVELET L SCHMIDT G TERRINE
Techniques numeriques de l'approximation variationnelle des problemes elliptiques
Tome 1 Institute Blaise Pascal Paris France April 1966 and Tome 3 March 1967

10 S M MORRIS W E SCHIESSER
SALEM—A programming system for the simulation of systems described by partial differential equations
Proc of Fall Joint Computer Conference December 9–11 1968 San Francisco California

11 C C TILLMAN
EPS—An interactive system for solving elliptic boundary-value problems with facilities for data manipulation and general-purpose computation
Department of Mechanical Engineering Massachusetts Institute of Technology June 1969

APPENDIX 1

The problem is to solve the two dimensional elliptic equation

$$\frac{\partial}{\partial x}\left(\frac{\sigma\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{\sigma\partial\phi}{\partial y}\right)=k$$

in which

$$\sigma = 10 + y$$

$$k = 0$$

for the following hollow quadratic field with the indicated Dirichlet boundary conditions

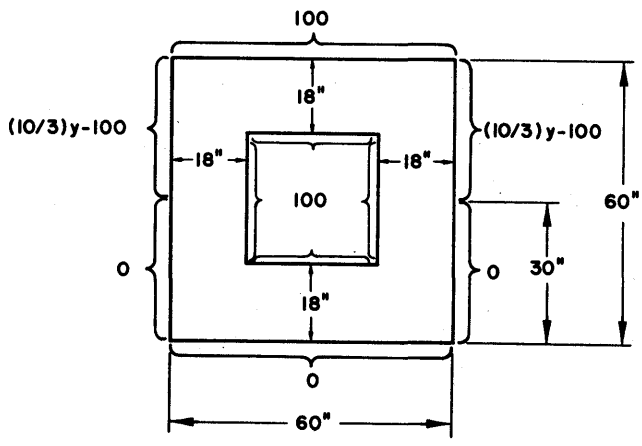


Figure 7—Appendix 1—Hollow quadratic field with the indicated dirichlet boundary conditions

```
THE SOLUTION CONVERGES WITHIN  5.00000E-02 IN  3.50000E+01 ITERATIONS, AND, FOR EACH  GRID POINT INDICATED, IT IS:

PN(0,0)= 0.000000E+00   PN(0,1)= 0.000000E+00   PN(0,2)= 0.000000E+00   PN(0,3)= 0.000000E+00   PN(0,4)= 0.000000E+00
PN(0,5)= 0.000000E+00   PN(0,6)= 0.000000E+00   PN(0,7)= 0.000000E+00   PN(0,8)= 0.000000E+00   PN(0,9)= 0.000000E+00
PN(0,10)= 0.000000E+00  PN(0,11)= 0.000000E+00  PN(0,12)= 0.000000E+00  PN(0,13)= 0.000000E+00  PN(0,14)= 0.000000E+00
PN(0,15)= 0.000000E+00  PN(0,16)= 6.666666E+00  PN(0,17)= 1.333333E+01  PN(0,18)= 1.999999E+01  PN(0,19)= 2.666666E+01
PN(0,20)= 3.333333E+01  PN(0,21)= 3.999999E+01  PN(0,22)= 4.666666E+01  PN(0,23)= 5.333333E+01  PN(0,24)= 5.999999E+01
PN(0,25)= 6.666666E+01  PN(0,26)= 7.333333E+01  PN(0,27)= 7.999999E+01  PN(0,28)= 8.666666E+01  PN(0,29)= 9.333333E+01
PN(0,30)= 1.000000E+02;
PN(1,0)= 0.000000E+00   PN(1,1)= 1.648782E+00   PN(1,2)= 3.056142E+00   PN(1,3)= 4.285074E+00   PN(1,4)= 5.375842E+00
PN(1,5)= 6.374485E+00   PN(1,6)= 7.277188E+00   PN(1,7)= 8.103919E+00   PN(1,8)= 8.848947E+00   PN(1,9)= 9.528312E+00
PN(1,10)= 1.015078E+01  PN(1,11)= 1.074819E+01  PN(1,12)= 1.137215E+01  PN(1,13)= 1.212682E+01  PN(1,14)= 1.322202E+01
PN(1,15)= 1.517424E+01  PN(1,16)= 1.948171E+01  PN(1,17)= 2.462341E+01  PN(1,18)= 3.010626E+01  PN(1,19)= 3.574936E+01
PN(1,20)= 4.147674E+01  PN(1,21)= 4.725468E+01  PN(1,22)= 5.306774E+01  PN(1,23)= 5.890904E+01  PN(1,24)= 6.477303E+01
PN(1,25)= 7.065380E+01  PN(1,26)= 7.654648E+01  PN(1,27)= 8.244455E+01  PN(1,28)= 8.833629E+01  PN(1,29)= 9.420142E+01
PN(1,30)= 1.000000E+02;
PN(2,0)= 0.000000E+00   PN(2,1)= 3.293151E+00   PN(2,2)= 6.104336E+00   PN(2,3)= 8.549493E+00   PN(2,4)= 1.074489E+01
PN(2,5)= 1.274132E+01   PN(2,6)= 1.455600E+01   PN(2,7)= 1.621376E+01   PN(2,8)= 1.771395E+01   PN(2,9)= 1.906610E+01
PN(2,10)= 2.028821E+01  PN(2,11)= 2.142877E+01  PN(2,12)= 2.257449E+01  PN(2,13)= 2.386230E+01  PN(2,14)= 2.550623E+01
PN(2,15)= 2.783543E+01  PN(2,16)= 3.123970E+01  PN(2,17)= 3.533116E+01  PN(2,18)= 3.981161E+01  PN(2,19)= 4.451247E+01
PN(2,20)= 4.934300E+01  PN(2,21)= 5.425684E+01  PN(2,22)= 5.923261E+01  PN(2,23)= 6.425769E+01  PN(2,24)= 6.932369E+01
PN(2,25)= 7.442571E+01  PN(2,26)= 7.955707E+01  PN(2,27)= 8.470536E+01  PN(2,28)= 8.985076E+01  PN(2,29)= 9.496361E+01
PN(2,30)= 1.000000E+02;
PN(3,0)= 0.000000E+00   PN(3,1)= 4.923736E+00   PN(3,2)= 9.114718E+00   PN(3,3)= 1.278053E+01   PN(3,4)= 1.608577E+01
PN(3,5)= 1.908646E+01   PN(3,6)= 2.183569E+01   PN(3,7)= 2.434825E+01   PN(3,8)= 2.662067E+01   PN(3,9)= 2.864191E+01
PN(3,10)= 3.042688E+01  PN(3,11)= 3.203278E+01  PN(3,12)= 3.356227E+01  PN(3,13)= 3.515943E+01  PN(3,14)= 3.700066E+01
PN(3,15)= 3.927964E+01  PN(3,16)= 4.213358E+01  PN(3,17)= 4.545555E+01  PN(3,18)= 4.909921E+01  PN(3,19)= 5.295115E+01
PN(3,20)= 5.693744E+01  PN(3,21)= 6.101681E+01  PN(3,22)= 6.516476E+01  PN(3,23)= 6.937357E+01  PN(3,24)= 7.364619E+01
PN(3,25)= 7.798208E+01  PN(3,26)= 8.237275E+01  PN(3,27)= 8.680057E+01  PN(3,28)= 9.123900E+01  PN(3,29)= 9.565315E+01
PN(3,30)= 1.000000E+02;
PN(4,0)= 0.000000E+00   PN(4,1)= 6.518893E+00   PN(4,2)= 1.207110E+01   PN(4,3)= 1.695580E+01   PN(4,4)= 2.136261E+01
PN(4,5)= 2.539424E+01   PN(4,6)= 2.911164E+01   PN(4,7)= 3.252622E+01   PN(4,8)= 3.561273E+01   PN(4,9)= 3.831885E+01
PN(4,10)= 4.062997E+01  PN(4,11)= 4.261277E+01  PN(4,12)= 4.439053E+01  PN(4,13)= 4.611749E+01  PN(4,14)= 4.794934E+01
PN(4,15)= 5.002168E+01  PN(4,16)= 5.241326E+01  PN(4,17)= 5.510130E+01  PN(4,18)= 5.801921E+01  PN(4,19)= 6.109875E+01
PN(4,20)= 6.428770E+01  PN(4,21)= 6.754419E+01  PN(4,22)= 7.085588E+01  PN(4,23)= 7.425222E+01  PN(4,24)= 7.774258E+01
PN(4,25)= 8.133390E+01  PN(4,26)= 8.501383E+01  PN(4,27)= 8.875769E+01  PN(4,28)= 9.253333E+01  PN(4,29)= 9.630133E+01
PN(4,30)= 1.000000E+02;
PN(5,0)= 0.000000E+00   PN(5,1)= 8.063937E+00   PN(5,2)= 1.494880E+01   PN(5,3)= 2.102695E+01   PN(5,4)= 2.653722E+01
PN(5,5)= 3.162348E+01   PN(5,6)= 3.636366E+01   PN(5,7)= 4.076709E+01   PN(5,8)= 4.475885E+01   PN(5,9)= 4.820812E+01
PN(5,10)= 5.101873E+01  PN(5,11)= 5.327888E+01  PN(5,12)= 5.516797E+01  PN(5,13)= 5.687274E+01  PN(5,14)= 5.855493E+01
PN(5,15)= 6.033391E+01  PN(5,16)= 6.227704E+01  PN(5,17)= 6.439279E+01  PN(5,18)= 6.665263E+01  PN(5,19)= 6.901524E+01
PN(5,20)= 7.142692E+01  PN(5,21)= 7.386238E+01  PN(5,22)= 7.633886E+01  PN(5,23)= 7.891363E+01  PN(5,24)= 8.162901E+01
PN(5,25)= 8.449255E+01  PN(5,26)= 8.748719E+01  PN(5,27)= 9.058156E+01  PN(5,28)= 9.373617E+01  PN(5,29)= 9.688436E+01
PN(5,30)= 1.000000E+02;
PN(6,0)= 0.000000E+00   PN(6,1)= 9.542446E+00   PN(6,2)= 1.773391E+01   PN(6,3)= 2.493435E+01   PN(6,4)= 3.153921E+01
PN(6,5)= 3.770349E+01   PN(6,6)= 4.354357E+01   PN(6,7)= 4.907079E+01   PN(6,8)= 5.415421E+01   PN(6,9)= 5.851381E+01
PN(6,10)= 6.178895E+01  PN(6,11)= 6.418735E+01  PN(6,12)= 6.602371E+01  PN(6,13)= 6.755752E+01  PN(6,14)= 6.897314E+01
PN(6,15)= 7.039014E+01  PN(6,16)= 7.187387E+01  PN(6,17)= 7.344437E+01  PN(6,18)= 7.508935E+01  PN(6,19)= 7.676619E+01
PN(6,20)= 7.843001E+01  PN(6,21)= 8.004845E+01  PN(6,22)= 8.163774E+01  PN(6,23)= 8.335192E+01  PN(6,24)= 8.527582E+01
PN(6,25)= 8.742158E+01  PN(6,26)= 8.976141E+01  PN(6,27)= 9.224995E+01  PN(6,28)= 9.481106E+01  PN(6,29)= 9.741640E+01
PN(6,30)= 1.000000E+02;
PN(7,0)= 0.000000E+00   PN(7,1)= 1.092357E+01   PN(7,2)= 2.028077E+01   PN(7,3)= 2.860552E+01   PN(7,4)= 3.626780E+01
```

Figure 8—Appendix 1—Tabular printout of the field potential for the two-dimensional elliptic problem.

This system could be a hollow square pipe carrying a fluid whose thermal conductivity is nonuniform, with the walls being subjected to the indicated temperatures.

With the understanding that PDEL solves equations by finite difference techniques in rectangular coordinates, the user of the language then has to indicate the following, in addition to specifying unambiguously the equation, parameters and conditions,

1. the rectangular grid to be used
2. the spacing between each grid point
3. the form of the printout

and assuming that the method used to solve two dimensional elliptic equations is the successive point overrelaxation, the following must also be specified,

4. the overrelaxation factor (if not specified, the optimum is used)
5. the error tolerance
6. the maximum number of iterations that are allowed

Assuming that the user specifies the following conditions,

1. a 30 by 30 grid
2. the spacing between each point in either direction is 1.0
3. the solution at each grid point is to be printed out, and also a discrete plot of the solution is desired
4. the overrelaxation factor is 1.70
5. the maximum error allowed is 0.05
6. the maximum number of iterations allowed is 100

then the following PDEL program is written to solve

the two dimensional elliptic field:

```
PDEL Program                                          Statement #
%  INCLUDE $PDEL(INITIAL);                                  1
   /*THIS PROBLEM SOLVES A 2 DI-                           2
     MENSIONAL ELLIPTIC EQUATION
     IN A HOLLOW QUADRATIC FIELD
     BY SUCCESSIVE OVERRELAXATION */
  % DECLARE (PARAM1, PARAM2) CHAR-                         3
     ACTER;
  % EQUATION = 'PARAM1 * PX, PX,                           4
     PHI + PARAM1 * PY, PY, PHI =
     PARAM2';
  % PARAM1 = '10 + Y'; % PARAM2 = '0';                  5, 6
  % DIMENSION = '2';                                       7
  % GRIDPOINTSX = '30';                                    8
  % GRIDPOINTSY = '30';                                    9
  % DELTAX = '2.0';  % DELTAY = '2.0';                10, 11
  % GEOMETRY = '(1:29, 1:8); (1:29, 22:29);                12
                (1:8 & 22:29, 9:21)';
  % BCOND = '(*, 0) = 0; (*, 30) = 100;                    13
              (0, 0:15) = 0; (30, 0:15) = 0;
              (0, 16:29) = (10/3) * Y - 100;
              (30, 16:30) = (10/3) * Y - 100;
              (9:21, 9) = 100;
              (9:21, 21) = 100;
              (9, 9:21) = 100;
              (21, 9:21) = 100';
  % MAXERROR = '.05';                                      14
  % ITERATE = '100';                                       15
  % ORF = '1.70';                                          16
  % PRINTINTX = '1.0';                                     17
  % PRINTINTY = '1.0';                                     18
  % PLOT = 'YES';                                          19
%  INCLUDE $PDEL(HEART);                                   20
```

Ahead of statement #1 of the program the user has to place the appropriate job control cards to give the operating system of the computer the necessary information about the job and where in secondary storage the PDEL translator is residing. The statements indicated play the following role:

1: calls from the computer system that part of the PDEL translator which performs all the necessary initialization. It must be the first statement in a program involving a new equation. $PDEL(INITIAL) is assumed to be the name of the data set where the initialization part of the translator is stored.
2: a comment statement
3: declares the names to be used for the parameters σ and k
4: defines the equation to be solved
5, 6: define the parameters σ and k
7: indicates that the field is 2 dimensional
8, 9: indicate that a 30 by 30 grid is to be used
10, 11: indicate that the distance between grid points is 2.0 and equal in either direction
12: indicates the geometry of the approximated field by specifying the points interior to the field
13: indicates the boundary conditions
14: indicates the maximum tolerable error in the solution
15: indicates the maximum number of iterations allowed
16: indicates the overrelaxation factor to be used
17, 18: indicate how often is space (grid point-wise, not space unit wise) the solution is to be printed out.
19: indicates that a scaled plot of the potential distribution, a contour plot, is to be printed out
20: calls from the computer system that part of the PDEL translator which performs the processing

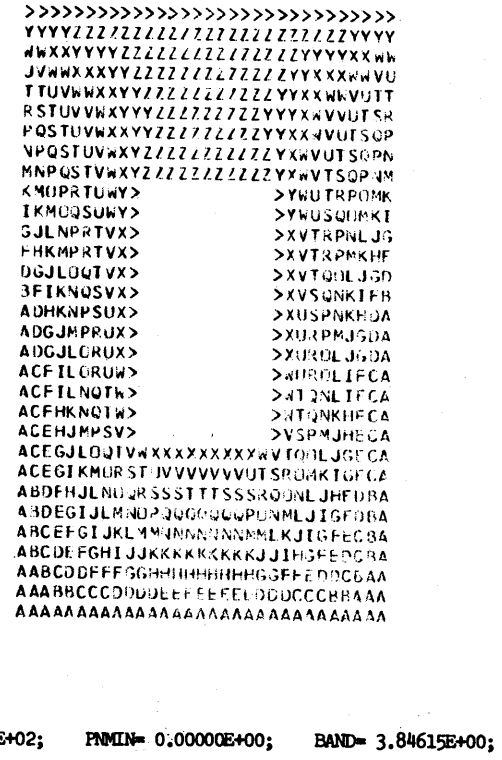


Figure 9—Appendix 1—Contour plot of the field potential for the two-dimensional elliptic problem.

of the PDEL program. $PDEL(HEART) is assumed to be the name of the data set where this key part of the translator is stored.

Solution printed out

The solution at each grid point for only a part of the field is shown.

In the contour plot, areas of smaller potential are represented by characters A, B, C, ··· and those of larger potential by X, Y, and Z. The grid point(s) with the largest potential is (are) represented by character >, while each letter represents a band (BAND) of potential equal to the difference between the maximum and the minimum potential in the field divided by 26 (the number of letters in the alphabet). The contour plot is elongated because the computer printer prints out characters leaving more space between each row than between each column.

SCROLL—A pattern recording language*

by MURRAY SARGENT III

University of Arizona
Tuscon, Arizona

INTRODUCTION

A number of routines have been developed recently to facilitate labeling of computer plotted output. One of the more versatile programs is that written by Freeman[1] which is capable of plotting characters in sequence including sub and superscripts, over and underscoring, using italics, changing fonts and returning to a saved coordinate. Programs of related nature have been written specifically for the purpose of text editing such as the IBM TEXT 360 and CALL 360 which are primarily printer oriented. Along these lines, all conversational programming systems have editing facilities. The routines share a common feature: the interpretation of character strings containing substrings specifying the desired output and other substrings specifying control functions. The substrings are separated typically by a break character such as a dollar sign or slash followed by a character representing the purpose of the substring. The use of a single character to represent a word or idea is as old as language itself (& for and, $ for dollar, etc.) and the characters so used are called logograms or logographs. One of the early programming languages to use logograms is APL although the purpose of that language is very different from the string languages involved above. A clear advantage of the logogrammatic language is its brevity. However this can be a confusing factor as well.

In this paper, we present a new logogrammatic language called SCROLL which extends the string language of Freeman to allow nesting of the subsuperscript, over-underscore and backward reference facilities and most important to include recursive procedure and measurement capabilities hitherto absent in plotting languages.* An abstract pattern can be defined by such a procedure and invoked as desired with specification of appropriate arguments to yield a specific pattern. Hence a mathematical fraction procedure measures the dimensions of the plotted output corresponding to its two arguments, one for the numerator and one for the denominator. This procedure then centers the numerator with respect to the fraction bar and denominator, while positioning the pattern elements vertically to prevent intersections. The arguments consist of any allowable sentences of the language and can in particular reference the procedure to which the arguments themselves belong. This allows one to draw, for example, a fraction in the numerator of another fraction.

The semantics and syntax of SCROLL are given in the next section of the paper together with numerous examples of plot output. A detailed discussion of procedures and measuring functions is given with examples in the third section. The utility of the language is discussed in the last section. In the appendices, SCROLL syntax is specified using a meta-language similar to that used in the COBOL report, and definitions of built-in SCROLL procedures given.

SEMANTICS AND SYNTAX

SCROLL sentences are composed of plot and control statements which, syntatically, can be mixed together in any order such that the final statement is a termina-

* Most of the work discussed herein was completed at Bell Telephone Laboratories, Inc., Holmdel, New Jersey.

* SCROLL is an acronym for **S**tring and **C**haracter **R**ecording **O**riented **L**ogogrammatic **L**anguage. The language has been incroporated into a general plotting system described in Bell Telephone Laboratories memorandum MM 69-1254-11 by M. Sargent III. Details of the SCROLL implementation can be found there.

TABLE I—Semantics of Logograms for IBM 360, CDC 6000 and Machine Independent Versions of SCROLL.

DEFINITION	LOGOGRAMS—Semantic 1 IBM 360	CDC 6000	LOGOGRAMS—Semantic 2 (machine independent)
Font change	0–9	0–9	0–9
Case Inversion	*A–Z*	*A–Z*	
Subscript	−	−	−
Superscript	+	+	+
Sub-superscript return	=	=	=
Italics	/ |	/ ∨	/ /
Under-overscoring	( )	( )	( )
Ending sentence	. ;	. ;	.
Carriage control	,	,	,
Column control	¢	=	*T* (FIV FORMAT)
Linewidth	#	[	*W* (*W*idth)
Character size	@	]	*S* (*S*ize)
Omitting output	¬ ¬;	¬ ¬;	*X*
Coordinate changes	< >	< >	*H* (*H*ere) *T* (*T*here)
Procedure calls	:	:	*C* (*C*all)
Diagnostics	?	↓	*D* (*D*iagnostics)
Plotting $	$	$	$
Changing semantics	!	↑	*L* (*L*ogogram)
Process statements	*	*	*
Null	%	=	*N* (*N*ull)
Backspace	&	∧	*B*

tion control statement.* A plot statement consists of any string of characters (a Hollerith literal) terminated by and not including a dollar sign. The action implied by a plot statement is the plotting of the characters constituting the statement. Hence the string 'ABC' means "plot 'ABC';." This statement cannot by itself constitute an entire sentence; it must be followed by a control statement which terminates the sentence.

Control statements are also character strings, but inevitably begin with a dollar sign and end according to context as described below. The first nonblank character following the dollar sign stands for the type of action demanded by the control statement. As such the character is a logogram and gives SCROLL its logogrammatic nature. More characters may be included in the statement depending on its purpose. Blanks occurring between the initial $ and the final character of the control statement are ignored unless they belong to a plot statement which is the argument of a SCROLL procedure or function call. The simplest SCROLL sentence consists of the single control statement '$.' which means "terminate the sentence." Combined with the plot statement above, one has the sentence 'ABC$.' which means "plot 'ABC'." Most control statements have fixed length. Those having variable length are terminated either by a per cent sign (%) of the $ of the next control statement.

In the remainder of this section, the control statements are defined and illustrated by figures containing prints of unretouched output obtained using an IBM 360/65 computer in conjunction with a Stromberg-Carlson 4060 microfilm recorder. The semantics given are for the IBM 360 version; Table I summarizes these semantics, the CDC 6000 version* and a machine independent set. Additional examples of SCROLL sentences are given in Appendix B which gives the definitions of the built-in procedures.

1. Specifying a new type font (see Figure 1a)
   $n change to the font numbered n ($1 \leq n \leq 9$). Four interpretive fonts** have been used herein: #1, the upper case English font; #2, the lower case English font; #3, the upper case Greek font, and #4, the lower case Greek font. The fonts include the special symbols . % + − = ′ ( ) / * \| $: , < > [ ] ¢ # & { } and characters representing integration, differentiation, infinity, summation, product and the Yale seal. Characters not

* The reader may prefer the schematic definition of SCROLL syntax given in the Appendix A to that given here.

* Debugged at the University of Arizona Computer Center.

** An interpretive font consists of a sequence of coordinates and delimiters which is interpreted and scaled to produce characters of desired size.

$n	A$4BC$0D$.	$A\beta\gamma D$
	A$3DHP$1E$.	$A\Delta\Theta\Pi F$
	A$2BCD$0EF$.	AbcdEF
$a	ABC.	AbC
$+	A$+QR$.	A^{QR}
$-	A$-QR$.	A_{QR}
$=	A$+R$=Q$.	$A^{R}Q$
	A$+R$-G=A$=B$.	$A^{R_{G}A}B$

Figure 1—Examples of SCROLL sentences illustrating control statements for (a) switching type fonts, (b) shifting case for one character and (c) sub and superscripting

$/	A$/BCD$.	*ABCD*
$|	A$/BCD$|EF$.	*ABCD*EF
$#	A$#4B$#3C$.	A**B**C
$($)+	A$(BCD$)+F$.	A$\overline{BCD}$F
$($)-	A$(BCD$)-F$.	AB$\underline{CD}$F
$<$>	$<'ABC$>.$.	:ABC
$$	P$2LOT $$$.	Plot $

Figure 2—Examples of SCROLL sentences illustrating control statements for (a) italics, (b) bold face, (c) over and underscoring, (d) returning to a saved plot position and (e) plotting a dollar sign

on key punch are retrieved by a number sign # followed by a key punch character. #A and #B, for example, yield { and } respectively. See the plot system memo for further discussion.

$0 return to previous font.

2. Shifting case for one character only (Figure 1b)
   $a where a is any letter of the alphabet inverts the shift for one character only: if $A is encountered and a lower case font is set up "A" will be plotted (instead of "a") and succeeding characters will be plotted in lower case.
   Note: The case may be shifted for one or more characters by changing to the appropriate font or by typing parts of SCROLL sentences in lower case when an upper case font is set up.

3. Subscripting and superscripting (Figure 1c)
   $− enter subscript mode,
   $+ enter superscript mode,
   $= return to previous sub or superscript mode.

4. Italics (Figure 2a)
   $/ enter italics mode and
   $| leave italics mode.

5. Under and overscoring (Figure 2c)
   $(remember where to start drawing a line, under or overscoring,
   $)+ or $)− overscore (+) or underscore (−) the characters between this $) and the corresponding $(; $(and $) are treated as a pair the same way right and left parentheses are treated in FORMAT statements,
   $) followed by anything else, draw a line between current plot coordinates and those saved by corresponding $(. See also $_ facility for drawing lines.

6. Ending sentence (see next section for examples)
   $. (or $;) If encountered in procedure or argument sentence, return to calling sentence; otherwise return to the plot system.

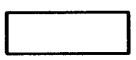
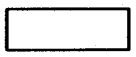
$,	The interpretation of $, starts one on the next line	
$@	A$@60B$@65C$.	A B C
$&	O$&/	Ø
$_	$_0,0;,2;6,2;6;$.	
	$_),2;6;,-2;-6$.	

Figure 3—Examples of SCROLL sentences illustrating control statements for (a) skipping to next line, (b) changing column, (c) backspacing and (d) drawing lines

7. Starting a new line (Figure 3a)
   $,n advance "carriage" according to value of n: n = plus, go to beginning of current line; n = 1, go to next frame; n = 0, skip a line; n = 2–9, advance (1/n)th of current frame; n = anything else, go to next line. If the control requests plotting below the frame, the frame is automatically advanced.

8. Specifying plot column (Figure 3b)
   $¢n (n= COLNO—variable in PLTPRM described in plot system memo) go to column n with respect to left side of film (ignore XORG and X, positioning variables). The variable COLNO determines the number of columns assumed on a frame and has the default value 80.0. Hence unless COLNO is reset larger than 100.0, the control sequence '$¢100' will require plotting outside screen boundaries and error messages will be issued. The letter n can be the name of a SCROLL process variable containing the desired column number. Hence on can set "tabs" for printing text.

9. Changing linewidth (Figure 2b)
   $#n for n = 1 to n = 4, use standard line density and change linewidth to n where width is proportional to 2**n (gives varying degrees of bold-face type); for n = 5 to n = 8 change linewidth to n − 4 and use light density.

10. Changing character size
    $@n change character size to SIZE*n/2, where SIZE is the nominal character size;
    $@0 go back to previous size.

11. Omitting output
    $¬ omit output until $¬; is encountered,
    $¬; restore normal output.

12. Saving and returning to a point (Figure 2d)
    $< save the current plot coordinates;
    $> go back to the coordinates saved by the $< corresponding to this $> ($< and $> are treated as a pair the way right and left parentheses are treated in FORTRAN FORMAT statements).

13. Calling and defining SCROLL procedures (see next section)
    $:an call the procedure named by the upper case alphanumeric character a and numeral n (1–9 or null),
    $:an(list) call the procedure named an with arguments given by list,
    $:(an, sentence) define a procedure named an by the SCROLL sentence sentence.
    $:? read characters on next card in FORTRAN input stream into SCROLL storage starting at this $ and continue interpretation. A 0–2–8 punch does the same thing and can occur anywhere in a SCROLL sentence.

14. Backspacing (Figure 3c)
    $&n backspace n ($1 \leq n \leq 9$) characters exclusive of control characters,
    $&0 backspace 10 plotted characters,
    $& followed by anything else—backspace one plotted character and process the next character as usual.
    Up to ten characters can be backspaced over at a given time unless fewer than ten positions have been established. To backspace more generally, one must use the $< and $> facility (see 12).

15. Rotating output
    $" [exp1][,exp2]% where exp1 and exp2 are SCROLL arithmetic expressions. The azimuthal angle (angle of rotation from the x axis in the x–y plane) is set equal to the value of exp1 if present, and the polar angle (angle of rotation from the z-axis towards the x–y plane) is set equal to the value of exp2 if it appears. The two angles have default values 0° and 90° respectively. Note that in SCROLL Version I, all previous measurement information is destroyed by this control statement. Hence one cannot, for example, draw an unrotated rectangle around a rotated pattern.

16. Shifting plot position and drawing lines (Figure 3d)
    $_field[_field] . . . % shift plot position and/or draw lines. A field contains one or more coordinate sets separated by semicolons and the sets themselves are SCROLL arithmetic expressions separated by commas. If a field consists of a single set, the plot position is shifted such that the first coordinate is added to the x position and the second to the y position. If a third coordinate appears it is used as the z (depth) coordinate in a projected drawing. If an equal sign precedes the x coordinate, the set specifies an absolute location on the plot screen. If more than one set appear in a field, lines are drawn between the

points they determine relative (no =) to the current plot position. If a '>' is the first character of a field, the coordinates are treated as vectors, that is, displacements from the current plot position. Lines are drawn *and* shifts occur. If a coordinate is omitted, it is assumed to be zero; if a single coordinate (no comma) appears, it is assumed to be x. Examples of shifting are given in another section and examples of line drawing in the "box" procedure. Shifts can be made in process stements also.

17. Writing diagnostic information
    - $?; terminate printing plot diagnostics,
    - $?n turn on the flag for diagnostic class n (see discussion of PLTDBG in next section of plot system memo for details),
    - $? followed by anything else causes descriptions of the type of actions resulting from subsequent control strings to be printed.

18. Plotting dollar sign (Figure 2e)
    - $$ plot dollar sign ($).

19. Changing SCROLL semantics
    - $!n use control semantics n (1 or 2), where semantics refers to the meanings of the logograms. $!1 causes the semantics described in this section to be used; $!2 causes semantics presumably specified by the user to be used (perhaps second set given in Table I).

20. Executing process statements (see Appendix B for examples)
    - $*...% execute the process statements given by the ellipsis (...) (see next section for further details).

21. Null statement
    - $% statement is ignored.

Further examples of SCROLL sentences

To demonstrate a little more of the power of SCROLL we consider the first equation in Figure 5. This resulted from interpretation of the sentence

'$:$T$($4Q0$.) = −$I[H, 4Q$]$−−$
= −$:$F$1(1$.2$.)[$C$.$Q$]$−+$.'

Here the string '$:$T$($4Q0$.)' is a procedure call (see next Section) to the time derivative procedure T which centers a dot above the plot output given by the argument, '4Q0.'. The argument itself is a SCROLL sentence in which '$4' switches to the lower case Greek font, 'Q' plots the letter ρ, '$0' returns to the previous font (the upper case English font) and '$.' ends the sentence. The next four characters '= −' plot an equal sign with blanks on either side followed by a minus sign. The '$$I$' plots i (the dollar sign inverts the case for I alone), '[H,' plots itself, '$4' switches to the lower case Greek font and 'Q]' plots ρ]. The '$ − −' subscripts a minus sign, '$=' returns to on-line mode, ' −' plots a minus sign with blanks on either side, '$:$F$1(1$.2$.)' calls the simple fraction procedure to plot the fraction one half, the '$$C$' plots Γ, '$−+' subscripts a plus and '$.' ends the SCROLL sentence. In the IBM implementation for fonts 1 through 4, [is given by #1 and] by #2.

The second equation resulted from interpretation of the sentence

'$:$B($:$P(A. B. N.) = $:$R
($:$F($A−$B$. B+Q−N=$=$.)$.)
$:$P(A. B. $N−1$.)$.)$.'

Here the "box" procedure B plots a rectangle tailored to fit plot output resulting from its argument. Similarly, the partial derivative (P), square root (R) and fraction (F) procedures shift their arguments into place and draw lines of appropriate length.

SCROLL PROCEDURES

In Figure 5 a complex interpretive font character is plotted namely the Yale seal.* One often wants to plot complicated groups of characters, such as a mathematical fraction and could, as for the Yale seal, define the desired pattern as an interpretive font character. In general this is an exceedingly tedious procedure. Instead one would like to define recurrent patterns by procedures and invoke particular patterns using appropriate arguments in the procedure calls. This extremely useful possibility is part of the SCROLL language. Specifically, the call is a control statement having one of the forms

(1) '$:an',

(2) '$:an(list)',

where a (any letter of the alphabet) and n (1–9 or null) identify the procedure, list is a SCROLL paragraph whose sentences comprise the procedure's arguments. The procedure facility is recursive, that is, a procedure

* The author is indebted to D. Barth for the coordinates of the Yale seal and other characters used herein for plotting. His Yale seal appears slightly differently in his article in *Computer Programs for Chemistry*, Ed. D. F. DeTar, Vol. 3, W. A. Benjamin, Inc., New York (1969).

$$\dot{\rho} = -i[H, \rho]_- - \tfrac{1}{2}[\Gamma, \rho]_+$$

$$\frac{\partial^n A}{\partial B^n} = \sqrt{\frac{A-B}{A^{C_n}}}\,\frac{\partial^{n-1} A}{\partial B^{n-1}}$$

Figure 4—Plot output resulting from interpretation of SCROLL sentences

can call itself, and procedures can define other procedures. Note that a SCROLL *procedure* differs from a SCROLL *function* in that the latter returns a value such as the length of a sentence as contrasted to the execution of plot and control statements.

The fraction procedure

In plotting mathematical equations one pattern which occurs frequently is the fraction. A procedure named F has been defined to plot a fraction.

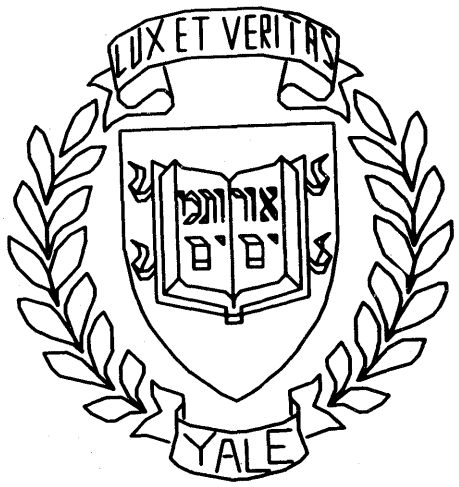


Figure 5—The Yale seal, example of a complex interpretive font character. Note that this figure is static: only its size can be changed. The SCROLL procedures allow one to define equally complicated figures which are dynamic, that is, the user can change parts of a figure merely by changing arguments to a procedure

It is called by the statement

'\$:$F(n, m)$'

which plots

$$\frac{n}{m}$$

where n and m are SCROLL sentences. In particular the call

'\$:$F(A - B\$. A\$ + C\$ - \$N\$ = \$ = \$.)$'

plotted the fraction within the square root in Figure 4.

Every SCROLL sentence constitutes a procedure definition. The nth argument in the procedure call is inserted whenever the characters '&n' ($1 \leq n \leq 9$) are encountered. For example, the fraction procedure F might be defined by the sentence

'\$(\$(\$_0.5S%&1\$) - \$)\$_ - 1.6S%&2\$_0.6S\$.'

Here the \$(saves the current x position for under or overscoring, the \$⟨ saves the x position for later reference, the \$_0.5S adds half a character size to the vertical (y) coordinate, &1 plots the first argument, \$) – underscores this argument, \$⟩ returns to the x coordinate saved by \$⟨, \$_−1.6S subtracts 1.6*SIZE from the vertical (y) coordinate, &2 plots the second argument, \$_0.6S restores the vertical coordinate to its value at the start of the call and \$. returns to the calling string. Simple fractions can be plotted by the procedure defined by

'\$+\$(&1\$) −\$&\$=\$−&2\$=\$.'

However, neither definition centers the numerator above the denominator or performs shifts to prevent intersection of lines. For this, one must use the process statement facility described in SCROLL *Process Statements* section.

Partial derivative and other procedures

An example of a procedure calling another procedure is the partial derivative procedure which plots

$$\frac{\partial^n y}{\partial x^n}$$

when called by the statements '\$:P(\$2Y\$. X\$. N\$.)'. Its definition is

'\$:F(#D\$+&3\$=&1\$. #D&2\$+&3\$=\$.)\$.',

where #D retrieves ∂ in fonts 1–4. Use of this procedure was made in drawing Figure 4.

An ordinary derivative procedure is available and is named D. A "time derivative" procedure named T

```
$:B5(SNOBOL$.)
$:B(ABC$.)
$:C($4AB$.)
$:D1(A)B$.)
$:H(PLTBCD$.)
$:A(5$.)
$:H(CALCUL$.)$:A(6$.)$:B(X=$4B$1$C$.))
```

(SNOBOL)

[ABC]

(αβ)

A)B

⟨PLTBCD⟩

⟨CALCUL⟩ → [X=βc]

Figure 6—Examples of SCROLL procedure calls illustrating the "bead" (oval), box, circle, diamond, hexagon and arrow procedures. The bead, box, circle, diamond and hexagon procedures produce figures of just the correct size to enclose the plot output resulting from their arguments

centers a dot above the character(s) specified by the argument. These and other procedures are built into the language (and plot system), are illustrated in Figures 6–8 and are defined in Appendix B. The user can define his own procedures simply by including

```
$:F($A$. A-B$.)
$:F1(1$. 2$.)
$:B1($:F(A$. A-B$.)$.)
$:B2($:F(A$. A-B$.)$.)
$:P1($:F(A$. A-B$.)$.)
$:D(A$. B$. $N$.)
$:P(A$. B$. $N$.)
```

$\frac{a}{A-B}$

$\frac{1}{2}$

$\left[\frac{A}{A-B}\right]$

$\left\{\frac{A}{A-B}\right\}$

$\left(\frac{A}{A-B}\right)$

$\frac{d^nA}{dB^n}$

$\frac{\partial^nA}{\partial B^n}$

Figure 7—Examples of SCROLL procedure calls illustrating the fraction and simple fraction procedures, the square bracket, curly brace and parenthesis procedures, the ordinary and partial derivative procedures. The fraction and bracketing procedures are constructed with precisely the correct sizes to fit the plot output resulting from their arguments

```
$:S($N=1$. N$.)
$:X($N=1$. N$.)
$:R($:F(A$.A-B$.)$.)
```

$\sum_{n=1}^{N}$

$\prod_{n=1}^{N}$

$\sqrt{\frac{A}{A-B}}$

Figure 8—Examples of SCROLL procedure calls illustrating the summation, product and square root procedures

statements of the form

'$: (an, sentence)'

in a call to the plot system. Here a and n identify the procedure and sentence is the SCROLL sentence comprising the procedure's definition.

SCROLL process statements

Particularly in the execution of procedures it may be necessary to calculate the dimensions of a substring or argument. For example, if the numerator of a fraction has different length than the denominator, the bar of the fraction should be as long as the larger of the two and the smaller should be centered with respect to the larger. Furthermore both should be shifted far enough away from the bar to prevent intersection. Clearly the dimensions of the numerator and denominator vary from call to call and cannot be successfully defaulted ahead of time. One needs a processing facility to calculate dimensions on the spot. The $* . . . % control statement serves this purpose and has the form

'$*statement[{, 1; }statement] . . . %'

Here "statement" can be

(1) a branch statement—
-)±n meaning branch to the present character position ±n,
-)n meaning branch to the nth position in current sentence, where n can be either an integer constant, e.g., 10, or a SCROLL variable (defined below) whose value has been assigned by a q: (see (2));) followed by anything else causes a return to the calling sentence (procedure or argument);

(2) an assignment statement

[q:] a [,b] . . . = expression

where the (optional) q: assigns the SCROLL variable q the current location counter (for later branching), where $a[,b]$ . . . can be SCROLL variables (see below), X, Y or Z which add the value of the expression to the current horizontal, vertical and depth plot positions respectively, nP which stores into the nth variable in the labeled common PLTPRM (see plot system memo, op. cit.), or 1, 2 or 3 which store the expression's value into the plot limit variables containing the x maximum, y maximum and y minimum respectively if that value exceeds the one already established (allows one to measure plot output dimensions quickly);

(3) a conditional statement

? logical expression {, statement} . . .

where the logical expression has a form described below and the statements are any process statements. The logical expression dominates all statements represented by the ellipsis. A null logical expression is true if and only if plotting is not suppressed.

SCROLL variables

SCROLL variables are named by single letters of the alphabet, A through W, and letters A–Z preceded by # (i.e., $\#A$). Those named by the letters A through W are local and automatic, that is, the values they assume in different procedures are unrelated and are lost upon return from the procedure in which they were set. Those named by $\#A$ through $\#Z$ are global and static, that is, their values are known to all procedures and can be modified in any procedure.

When a SCROLL variable is used in an arithmetic context, it is assumed to be floating-point. When used in a logical context, a non-zero value is interpreted as true; a zero value as false.

A SCROLL arithmetic expression consists of a single arithmetic primary, a unary arithmetic operator followed by an arithmetic primary or two or more such expressions separated by binary arithmetic operators. An arithmetic primary can be a fixed point constant, e.g., 0.5 or 3, a SCROLL variable, a variable in common PLTPRM (nP retrieves the nth variable), the letters X, Y, Z which return the current x, y and z coordinates, a function reference or an arithmetic expression. In arithmetic context, fixed-point constants and SCROLL variables are assumed to have floating-point format. Values of SCROLL variables and fixed point constants have units of the plot screen. However if a fixed point constant is followed by the letter S, it is multiplied by the current character size before being used. There are four binary arithmetic operators, + (addition), − (subtraction), * (multiplication) and / (division). There are two unary operators, + (used for emphasis only) and − (negation). Binary operators must occur between two primaries or between a primary and a unary operator which precedes a primary.

Expressions are evaluated left to right subject to the following hierarchy. Functions are evaluated first, negation (unary −) second, multiplication and division third, and addition and subtraction last. The unary plus is ignored. Primaries consisting of parenthesized expressions are evaluated before arithmetic operations are performed and can be used as in algebra to override the hierarchy.

Examples of legal arithmetic expressions are

$$A*B/(C-D)$$
$$F+E(A\$-Q\$=X\$.)*-D$$
$$B(\$1\$-Q\$.)$$

The second and third examples contain function references described below. Note that $A*-D$ is a legal expression, for the minus sign is interpreted as a unary operator signifying arithmetic negation rather than the binary operator indicating subtraction. Examples of illegal expressions are

$AB-C$	either an operator is missing between A and B, or SCROLL variable is misnamed (only one letter is allowed),
$(A*B*(C+D)$	unpaired parenthesis,
$A-*B$	two binary operators must be separated by a primary.

A SCROLL logical expression consists of a single logical primary, a logical negation operator followed by a logical primary or two or more such expressions separated by binary logical operators. A logical primary has the value true or false and can be a fixed-point constant as above, a SCROLL variable, a logical expression enclosed in parentheses or two arithmetic expressions separated by a relational operator. In logical context, fixed-point constants and SCROLL variables are considered true if and only if they are nonzero. Zero values are false. The relational operators are > (greater than) < (less than), >= (greater than or equal to), <= (less than or equal to), = (equal to), and ¬= (not equal to). The binary logical operators are | (logical or) and & (logical and). There is one unary logical operator, ¬ (logical negation). The total hierarchy of operations for logical expressions is as

follows

Operation	*Precedence*
Evaluation of functions	8 (highest)
Unary + and −	7
* and /	6
+ and −	5
>, <, >=, <=, = and ¬=	4
¬	3
&	2
|	1 (lowest)

As with arithmetic expressions, parentheses can be used to override the hierarchy. Examples of valid logical expressions are as follows

$$A>=B(ABC\$.) \mid \neg C$$
$$F$$
$$G\&(A \mid B)$$

Process functions

Five functions are defined for the measurement of the plotted output resulting from SCROLL sentence interpretation: B returns the bottom and length of its argument; D returns the length, height, bottom difference height-bottom of its argument; E returns the length alone, G is the same as D except that plotting can take place concurrently with measurement, and H returns the height and length of the argument. When used in an arithmetic statement other than a simple assignment, only one value (the first) is returned; in simple assignment statements, the first value returned is stored in the variable indicated, the second in the next variable in the alphabet, etc. The dimensions are all given in plot screen units. Vertical measurements (height and bottom) are made relative to the bottom of an on-line (not sub-superscripted) period. The functions are recursive.

In addition a maximum function M is defined which returns the maximum value given by the SCROLL variables appearing as arguments (see example below).

Example of use

As an example of process statement use, consider an extended definition of the fraction procedure F

```
'$*  A=B(&1$.); C=H(&2$.); C=C+0.2;
                                  E=M(B, D);
     $__,.5S$<$<$__(E-B)/2, -A+.2
     %&1$>$__0; E,__(E-D)/2,-C
     %&2$>E,-.5S$.'
```

Here for the sake of readability, numerous blanks have been included; ordinarily one deletes blanks to save space and execution time. The first statement stores in A and B the bottom and length of the first argument (the numerator), the second statement stores in C and D the height and length of the second argument (the denominator) and the third statement adds 0.2″ to C. The fourth statement stores in E the larger of the lengths stored in B and D. The control statement $\$\_\_,.5S$ in line 2 shifts the plot position up one half a character size. The two $\$<$ statements save the plot position for future reference and the $\$\_\_(E-B)/2, -A+.2S$ shifts the plot position so that the numerator will be centered with respect to the length E and the bottom of the numerator will be 0.2 of a character size above the bar of the fraction. The percent sign in line 3 terminates the shift field, and &1 plots the numerator. The $\$>$ then returns to the plot position established before the numerator was shifted into place and the $\$\_\_0;E,\_\_(E-D)/2,-C$ draws the bar of the fraction (length E) and then shifts so that the denominator is 0.2 of a character size below the bar of the fraction. The percent sign in line 4 terminates the shift field, &2 plots the denominator, $\$>$ returns to the saved plot position and $\$\_\_E,-.5S$ shifts to the original vertical position and a horizontal position on the right of the fraction. Note that by saving and returning to a known reference position, one does not have to know where the numerator and denominator finish.

CONCLUSIONS

The SCROLL language has been used extensively in the preparation of lecture slides such as that printed

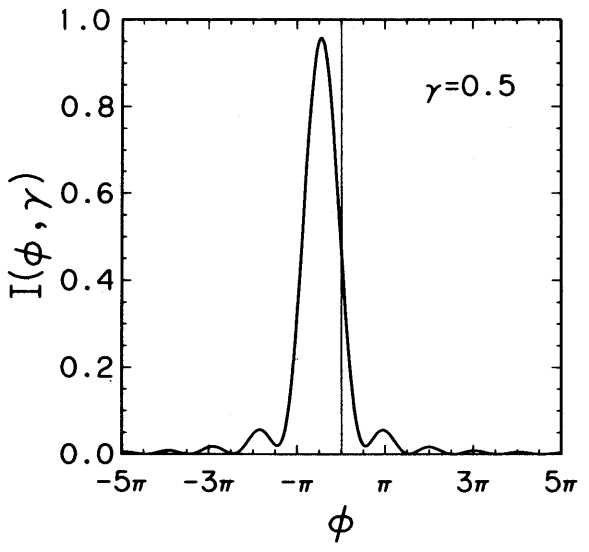


Figure 9—A graph labeled using simple SCROLL sentences

in Figure 4 and in the labeling of graphs for publication as shown in Figure 9. The cost involved is dramatically less than that incurred when a draftsman is used. Specifically, the first equation if Figure 4 ran on an IBM 360/65 (using FORTRAN IV level *G*) 0.226 seconds; the second for 0.64 seconds and the complete labeled graph in Figure 9 for one second. A second costs about ten cents on a typical IBM 360/65, and the cost of a frame on the Stromberg-Carlson 4060 microfilm recorder varies between eight and sixty cents according to load. This yields a cost for either slide of about thirty cents. Furthermore, the turn around time for the slides is generally less than a day. Compared to a draftsman, this is orders of magnitude cheaper and faster. Inasmuch as SCROLL with its recursive procedure and measurement capabilities now provides the user with the power hitherto only available from a draftsman, it is felt that SCROLL represents an important advance in the preparation of figures. It is felt that in general SCROLL represents a substantial advance in the computer preparation of manuscripts for publication.

ACKNOWLEDGMENTS

The author is indebted to numerous colleagues at Bell Telephone Laboratories for helpful discussions. In particular, he would like to thank A. G. Fox, C. T. Schlegel, D. A. Vaughan and D. P. Allen for editorial suggestions and J. F. Gimpel and I. P. Polonsky for illuminating conversations about programming languages. The author is especially grateful to R. E. Griswold for helpful suggestions of all kinds and for finding numerous errors in the SCROLL implementation, to G. C. Freeman of the Yale Computer Center whose MAP-coded plotting routines served as starting points for some of the principal subprograms, to J. D. Beyer for very helpful discussions about syntactic analysers and to D. Barth of Yale University for the coordinates of most characters in the interpretive type fonts.

APPENDIX A—SCROLL SYNTAX

In this appendix, a formal definition of SCROLL syntax is given in terms of a meta-language much like that used in the IBM PL/I Reference Manual (C28–8201).

A.1 The Syntax Meta-language

The meta-language used below to define the syntax of SCROLL consists itself of literals, variables, expressions and operators much as SCROLL or any other language does. More specifically, a literal consists of any character which is preceded by a bar (|), a blank, a left square bracket ([) or a left curly brace ({) and is followed by a bar, a blank, a right square bracket (]) or a right curly brace (}). A variable is named by a lower-case letter of the English alphabet followed by any non-zero combination of such letters and underscores (_). A primary is a constant or a variable or a bracketed expression. Square brackets are used when the expression is optional; curly braces when required. A unit is a primary optionally followed by an ellipsis (...), the latter indicating that the primary is repeated zero or more times. An expression is one of three things: 1) a single unit, 2) a unit followed first by one or more blanks and then by another expression, or 3) a unit followed first by a bar (1) optionally preceded and followed by blanks and then by another expression. In the second case the values of the unit and expression are concatenated; in the third case they are considered to be alternatives. A variable is given the value of an expression when its name is followed first by a colon (:), then by one or more blanks and finally by the expression. A variable is given the value of an English phrase when the variable is followed by a colon, then one or more blanks and finally by the phrase. A phrase is distinguished from an expression in that the former contains undefined variables and always makes sense as a phrase. When a syntactic symbol | []{} is to be used as a literal, it is underlined. Hence ⊥ is a literal | rather than the operator separating alternatives.

In particular the syntax of this meta-language is defined in terms of the language itself as follows:

l_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
literal:	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| l_letter |⊥|[|]|{|}|0|1|2|3|4|5|6|7| 8|9|,|:|;|'|"|?|+|−|*|/|<|>|(|)|%| =|$|!|¢|@|&|#|_
variable:	l_letter {l_letter | _}···
primary:	literal | variable | [expression] | { expression }
unit:	primary [···]
expression:	unit [[blank]··· {⊥|blank} [blank]··· unit]···
definition:	variable : [blank]··· expression | an English phrase

A.2 SCROLL Syntax

digit:	0|1|2|3|4|5|6|7|8|9
integer:	digit···

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

character: letter|blank|digit|⊥|+|−|*|/|=|.|,|¬|<|>|(|)|?|!|;|:|'|%|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|&|¢|"

simple_logogram: digit|letter|+|−|=|/|⊥|·|;|$|<|>|(|%|¬

delimiter: % | null if a control statement follows

constant: integer [·] | [integer] · integer

variable: A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|#|letter

factor: constant|variable|X|Y|Z| integer P

function: letter (sentence) | letter (variable [, variable]···)

a_primary: factor | (a_expression) | function

a_term: [+|−]··· a_primary

a_expression: a_term [{+|−|*|/} a_term]···

relation: a_expression {>|<|=|<=|>=|¬=} a_expression

l_primary: factor | (l_expression) | relation

l_term: [¬]··· l_primary

l_expression: l_term [{&|⊥} l_term]···

expression: l_expression | a_expression

angle_set: a_expression | [a_expression] , a_expression

coordinate_set: [=][a_expression][, [a_expression]] [, a_expression]

field: [coordinate_set ;]··· coordinate_set

procedure_name: letter [1|2|3|4|5|6|7|8|9]

l hs: integer P | variable | X | Y | Z

process_statement: >[[*{+|−}]{integer|variable}]| [variable:]··· lhs [, lhs] = expression | ? l_expression {, process_statement}···

control_field:
- simple-logogram |
- , [blank|digit|+] |
- ¢ {integer|variable} |
- " angle_set delimiter |
- # {0|1|2|3|4} |
- @ {0|1|2|3|4} |
-) [+1−] |
- & [digit] |
- ? [digit | ;] | ! {1|2} | ¬ ; |
- : procedure_name [(paragraph)] |
- : (procedure_name , sentence) | : ? |
- _ field [_ field]···delimiter |
- * process_statement [; process_statement]··· delimiter

control_statement: $ control_field

plot_statement: {character | # {character | #}}···

sentence: [plot_statement | control_statement]··· $.

paragraph: sentence [[blank] sentence]···

APPENDIX B—PROCEDURE DEFINITIONS

In this appendix, the built-in SCROLL procedures are defined. The name of the procedure is given first, then the call indicating the number of arguments for the procedure and a brief description of the procedure and finally the definition itself.

1. ALIGN—$:A9(&1 &2 &3), where &1 is to be centered below &3 and &2 is to be centered above &3 (used for summations and products).

```
'$* A=D(&3$.) $-$* D=D(&1$.);
                G=D(&2$.);J=M(A,D,G) $=$
 *C=C-E-.2;B=B-I+.2;?¬,X,1=J,
   2=B+H,3=C+F,> $<$<$<$_(J-A)/2
%&3$-$>$_(J-D)/2,C %&1$>$_(J-G)/2,B
                %&2$=$>$_J$.'
```

2. ARROW—$:A(&1), where an arrow of length &1 is to be drawn. Plotting terminates at the arrow's tip; the arrow is horizontal pointing to the right if &1 is positive and to the left if &1 is <0.

```
'$* A=&1;B=.5;?A>0,B=-B;
                $_>A_;B,.3_;B,-.3$.'
```

3. ARROW1—$A1:(&1), where an arrow of length &1 is to be drawn. Plotting terminates at the arrow's tip; the arrow is vertical pointing up if &1 is positive and pointing down if &1 is negative.

```
'$* A=&1;B=.3;?A>0,B=-B;
                $_>,A_;.5,B_;-.5,B$.'
```

4. BEAD—$:B5(&1), where &1 is to be enclosed within a bead (oval).

```
'$* A=D(&1$.);D=D+.4;2,H=D/2;
                1,E=A+D;?¬,3=-H,X=E,>;
 I=14P;14P=D $<$_,-H$⟨$@2#E$⟩$_H
   $⟨$_,D;A,D_⟩A%#F$⟩$@0$* 14P=I;Y=
H-(B+C)/2%&1$>$_E$.'
```

5. BOX—$:B(&1), where &1 is to be plotted with a box (rectangle) around it.

```
'$* A=D(&1$.);H=(B+C)/2;2,F=H-
                C+.4;1,A=A+.8;?-,3=-F,
X=A,>.$_,-F;,F;A,F;A,-F;,-F$<$_.4,
                -H%&1$>$_A$.'
```

6. BRACES—$:B2(&1), where &1 is to be enclosed in curly braces.

'$:B9(&1$.#A$.#B$.)$.'

7. GENBRK—$:B9(&1 &2 &3), where &1 is to be enclosed in the bracketed pair given by &2 and &3.

'$* A=D(&1$.);1,H=A+D/2+.6;?¬,
2=B+D/10,3=C−D/10,X=H,>;
I=14P; 14P=D; E=D/4+.2 $<$<$_,C$<
$@2&2$>$_H−E%&3$@0$*14P=I$>$_E&1
$>$_H$.'

8. BRACKET—$:B1(&1), where &1 is to be enclosed in square brackets.

'$* A=D(&1$.); E=D/6+.1; 1,A=A+2
*E+.6; 2,B=B+.2;3,C=C−
.2; ?¬,X=A,>$<$_E,C;,C;,B;E,B_A−E,B;A,B;
A,C;A−E,C_E+.3%&1$>$_A$.'

9. CIRCLE—$:C(&1), where &1 is to be encircled.

'$* A=D(&1$.); D=M(A,D); 1,D=1.4*D
+.4; 3, E=−D/2; ?¬,
2=−E,X=D,>; I=14P; 14P=D $<$<$−,
E$@2#0$@0$>$* 14P=I; X=(D−A)/2;
Y=−(B+C)/2%&1$>$_D$.'

10. ORDINARY DERIVATIVE—$:D(&1 &2 &3), where the &3th derivative of &1 with respect to &2 is to be plotted.

'$:F(#8$+&3$=&1$. #8&2$+&3$=$.)$.'

11. DIAMOND—$:D1(&1), where &1 is to be enclosed within a diamond.

'$* A=D(&1$.); I=.75*D+A/2+.4;
2,J=2*I/3; 1,E=I+I; ?¬,
3=−J,X=E,> $<$_;I,J;E;I,−J;_I−A/2,
−(B+C)/2 %&1$>$_E$.'

12. SIMPLE FRACTION—$:F1(&1 &2), where the ratio of &1 to &2 is to be plotted. No measurement of arguments is performed: they should have equal widths, no undershoots and full height.

'$($+$(&1$)−$=$)$−&2$=$.'

13. GENERAL FRACTION—$:F(&1 &2), where the ratio of &1 to &2 is to be plotted. The arguments can have different dimensions.

'$* A=D(&1$.); E=D(&2$.); I=M(A,E);
?¬,X,1=I, 2=D+.7,
3=.3−H,> $_,.5$<$<$_(I−A)/2,−C
+.2%&1$>$_0;I_(I−E)/2,−F−.2%&2$>$_I
,−.5$.'

14. GRID—$:G(&1 &2 &3 &4), where a rectangle is to be drawn containing &1 by &2 boxes each with length &3 and height &4. Plotting starts and terminates at the upper left-hand corner of the grid.

'$* E=&3; H=&4; 1, A=&1*E;
3,B=−&2*H; ?¬,>; I,J = 0 $_,J;
A,J $* J=J−H; ?B−J>.1,>−31$_I;I,B
$* I=I+E; ?I−A>.1,>−27$.'

15. HEXAGON—$:H(&1), where &1 is to be enclosed within a hexagon.

'$* A=D(&1$.); 2,H=D/2+.4; F=H/3;
A=A+.8; 1,G=2*F+A; ?¬,
3=−H,X=G,>$<$_>F,H;A,;F,−H;−F,−H;
−A,;−F,H_F+.4,−(B+C)/2%&1$>$_G$.'

16. INTEGRAL—$:I(&1 &2 &3), where &1 gives the lower limit of integration, &2 the upper limit and &3 the variable of integration.

'$@4#3$@0$<$_−.8,−.8$−&1$>$_
−.3,1.8&2$=$_,−1#8&3$.'

17. MIDDLE—$:M(&1 &2), where &2 is to be centered in the box &1 characters long. One starts in the middle of the left side of the box and ends in the middle of the right side.

'$* E=&1; ?¬,X=E,>; A=D(&1$.);
X=(E−A)/2;Y=−(B+C)/2 %&2$>$_E$.'

18. PAREN—$:P1(&1), where &1 is to be enclosed in parentheses.

'$:B9(&1$. ($.)$.)$.'

19. PARTIAL DERIVATIVE—$:P(&1 &2 &3), where the &3th partial derivative of &1 with respect to &2 is to be plotted.

'$:F(#D$+&3$=&1$. #D&2$+&3$=$.)$.'

20. SQUARE ROOT—$:R(&1), where the square root of &1 is to be plotted.

'$*A=D(&1$.);A=A+.4;E=.2$_,
C+D/2_>E,.1;D/12+E,−D/2−E−.1;
D/6,D+2*E_;A;A,−E_E,−B−E$*?¬,
X=A,>$<&1$>$_A$.'

21. SUMMATION—$:S(&1 &2), where the summation of &1 to &2 is to be plotted. Note that &1 must contain the equal sign if desired, e.g., if one wants A=1 below the sigma, &1 should equal 'A=1$.'.

'$:A9(&1$. &2$. #6$.)$.'

22. TIME DERIVATIVE—$:$T$(&1), where a dot is to be plotted above the argument.

```
'$* A=G(&1$.);I,X=(A-E(.$.))/2;
                    Y,E=B+.2%.$_I,-E$.
```

23. PRODUCT—$:$X$(&1 &2), where the product (Π) is to be plotted from &1 to &2. &1 must contain the equal sign if desired as for summation.

```
'$:A9(&1$. &2$. #9$.)$.'
```

AMTRAN—An interactive computing system

by JURIS REINFELDS, NIELS ESKELSON, HERMANN KOPETZ and GERHARD KRATKY

University of Georgia
Athens, Georgia

INTRODUCTION

The AMTRAN system (**A**utomatic **M**athematical **TRAN**slator) is a multiterminal conversational mode computing system which enables the mathematically oriented user to interact directly with the computer in a natural mathematical language. The first version of AMTRAN was developed at the George C. Marshall Space Flight Center in Huntsville,[1] and was implemented on IBM 1620 and 1130 computers and, as a time-sharing version, on a Burroughs 5500 computer. A modified 1620 console version is currently in use at the University of Georgia.

In connection with the project of implementing a multiconsole version on an IBM 1130 computer, the AMTRAN language has been revised and formally defined at the University of Georgia Computer Center.[2,3]

The following objectives have been of primary importance in the development of the AMTRAN systems:

First, the initial use of a computer by the mathematically oriented nonprogrammer and scientist should be made easy by using a simple language as similar to mathematical notation as possible. The language should be designed for incremental learning so that a user may successfully use the system without knowing all of its details.

Second, the system should provide powerful programming capabilities for the solution of medium and large scale problems and complicated algorithms by the more experienced user or professional programmer.

Third, since AMTRAN was conceived as a special purpose language for mathematical and scientific use, the system should provide more flexibility in programming, debugging, and turnaround time compared to conventional computing systems.

Fourth, the new design and definition of the AMTRAN language should have as few restrictions, exceptional rules to remember, and departures from the well-known semantics of algebra as possible without reducing the power of the system. The system should be fully recursive and there should be no practical limitation to the length of variable and program names, the number of defined variables, the dimensions of arrays, etc.

DEFINITION OF THE PROGRAMMING LANGUAGE AMTRAN

The BNF notation is the only widely used formal method for the definition of a computer language. It has been derived from strictly structural considerations. The use of bracketed English words as metasymbols has made the notation look less formidable but, at the same time, has introduced semantic aspects which originally were not present. As long as these semantic aspects served only to characterize necessary structural categories, the method was as originally intended. As soon as strict semantic categories were established, with no structural characteristics, difficulties arose. This can be explained by a simple example. The ALGOL 60 report[5] states:

⟨variable identifier⟩ : : = ⟨identifier⟩

⟨simple variable⟩ : : = ⟨variable identifier⟩

The introduction of the categories ⟨simple variable⟩ and ⟨variable identifier⟩ is necessary to distinguish between a strictly formal α-numeric string (⟨identifier⟩) and a special type of a variable. This distinction is based only on the meaning assigned to a particular string; the structure remains the same. Therefore, the difference is not really expressed by the above method since the distinction between the categories is left to the arbitrary interpretation of the metasymbols ⟨identifier⟩, and ⟨simple variable⟩ and is by no means formally defined. Omitting all these questionable 'semantic categories' would diminish the content of such a language definition considerably.

Realizing these difficulties, a new general method of formal definition of a mathematically oriented computer language was developed at the University of Georgia Computer Center.[2] By introducing different levels of the language, one structural and several semantic levels, it is possible to distinguish between the structure of a language and the meaning attached to the structural elements. A large part of the semantics is systematized in notions like type, range, sign, dimension of numerical quantities, and binding power of mathematical operators. It is found to be sufficient to introduce a few well-defined semantic values.

Each structural element is assigned a semantic and dimensional characteristic which carries the information associated with the structural unit. A structural unit and its semantic and dimensional characteristic are thus combined to form a constituent of the language. The idea of the new method of definition is to describe the language by setting up production rules for constituents[2] rather than for structural units by themselves. This systematization and formalization covers a much larger part of the language than mere 'syntax' definitions or the definition by BNF notation. Now it is possible to resolve the previous ALGOL example. The structure of ⟨simple variable⟩, ⟨variable identifier⟩, and ⟨identifier⟩ are the same, an alphanumeric string. But, since the semantic characteristics of a strictly formal alphanumeric string and of a simple variable are different, they form different constituents of the language. Therefore, the distinction between these categories is not left to the interpretation of the reader but is a part of the language definition.

BASIC FEATURES OF AMTRAN

Language features

The complete definition and a detailed description of the AMTRAN language appears in References 3 and 6. In this chapter, we shall describe the basic features of the language in accordance with the design principles and the aims of the AMTRAN systems as presented in the introduction.

Basic operators and functions

+, −, *, /, power, unary minus, SQRT, LN, ABS, EXP, LOG, SIN, COS, TAN, TANH, ARCTAN.

These operators are intrinsic to the system and, by using the well-known semantics of algebra and the familiar names for the functions, they can be used immediately by the nonprogrammer.

Absence of dimension statements

Arrays are created and changed with complete freedom at run time.
Examples:

$$X = \text{ARRAY } (0, 5, 5)$$

creates an array X with the values

$$0, 1, 2, 3, 4, 5$$

$$Y = X + 1$$

creates an array Y with the values

$$1, 2, 3, 4, 5, 6$$

even if Y had been a scalar or an array with another dimension before. Another operator to construct arrays is the concatenate operator &.

$$X = 0 \,\&\, 1 \,\&\, 2 \,\&\, 3 \,\&\, 4 \,\&\, 5$$

creates the same array as array X in the previous example.

Absence of declaration statements

The type of a variable is automatically defined through the assignment statement until it is changed by another assignment statement. At execution time, the control routine for each operator checks the type and range, sign and dimension of the operands.

A new concept is used for the handling of integers. They are stored and treated internally as real numbers, but a special rounding routine preserves their integer status through any arithmetical manipulations. Every time an operator requires an integer argument, the system examines the real representation of the value of the operand to determine whether it represents an integer number; an error message is typed if it does not. Thus, AMTRAN will give the right results for $(-2.5)^2$ as well as for $(-2.5)^{3*\text{SIN}(\pi/6)+0.5}$.

Automatic array arithmetic

The basic operators and functions mentioned in 3.1.1. and the relational operators can be used not only for scalars, but also on arrays. Thus, the user may compute directly with the numerical representations of functions without writing loops.
For example, the function

$$y = \frac{2}{\sqrt{\pi}} e^{-x} \sin x$$

is represented in AMTRAN as

$$Y = 2/\text{SQRT } PI*\text{EXP} - X*\text{SIN } X$$

where X can represent an array of 100 equally spaced intervals generated by X = ARRAY (0, PI, 100). The resulting function Y is represented by an array of 101 numbers, where each Y-value is the value of the above function for the corresponding X-value.

Conditional operators: IF, THEN, ELSE

Relational operators: GT, GE, EQ, LE, LT

Boolean operators: NOT, AND, OR

They are basically the same as in ALGOL 60 except that each IF has a corresponding FI (ALGOL 68 style) at the end of the conditional expression to avoid the dangling ELSE problem.

Unconditional branch and loop

The GO TO operator can be used for transfer of control to any numbered statement in a program. The argument of GO TO may be any expression which returns a scalar value. Non-integer values cause a warning message.

The REPEAT-operator is used to repeat a group of statements a specified number of times. These repeat-loops can be nested arbitrarily.

Generality of operands

As a general rule, every operand or parameter in AMTRAN can be an expression, but the result of this expression has to fulfill the semantic requirements its operator asks for. Example: The third parameter of the ARRAY operator (number of intervals) may be an expression, but the result has to be an integer with the dimension one.

Fully recursive programming capabilities

An example of an inherently recursive function is Ackermann's function $A(M, N)$, defined over the positive integers and zero:

1. M = IN 1, N = IN 2
2. A = IF M EQ 0 THEN $N + 1$ ELSE IF N EQ 0 THEN $A(M - 1, 1)$ ELSE $A(M - 1, A(M, N - 1))$ FI FI
3. NAME A

Statement 1 picks up two arguments which have to be provided by the program call. For example, $A(2, 4)$ will give $M = 2$ and $N = 4$ upon execution of statement 1.

Powerful instruction set which can easily be expanded by the user

The philosophy of AMTRAN is that the user should be given a powerful basic set of instructions which are intrinsic to the system, together with a disc file library of instructions which are actually routines written in AMTRAN. In addition, the user can define his own high level operators by writing special AMTRAN routines. Operators for automatic numerical analysis (integration, derivation), satisfactory for routine situations, are also included.

ASK- and TEACH-operators

The ASK-operator can be used to program a dialogue between the computer and the user. If the user is not satisfied with a system message, he can react with 'ASK,' and the system will respond with a more detailed message. This feature makes AMTRAN to a truly conversational system. The experienced user does not have to spend time running through the questions and answers, which are of great importance for the average and beginning AMTRAN-user.

The TEACH-program allows the user to learn how to use AMTRAN directly on the console. The new AMTRAN-user does not have to take a course in programming; he need not study a programming language or learn how to read computer outputs. He can get started with a simple teach program on the console in a few minutes without having to learn complicated rules. If there occurs a problem in using AMTRAN, the user can use the TEACH-operator and run the part of the teach program which refers to his problem.

Call by symbol concept

The call by symbol concept allows the passing of executable strings as parameters to subprograms. This symbolic expression can contain variables local to the calling program and variables local to the subprogram. Everytime the parameter is invoked within the subprogram, the symbolic expression is evaluated using the actual internal and external variables.

Three modes of operation

1. Execute mode: An interactive system must have an execution mode (or desk calculator mode) where each statement is executed immediately and control is returned to the keyboard. This is the default mode in AMTRAN.

2. Suppressed mode: The suppressed mode (delayed mode) allows the user to construct programs which are syntax checked and stored for execution at a later time.

3. Checking mode: AMTRAN has a third mode, the checking mode, which allows the user to execute parts of suppressed programs while they are being constructed. This is an important aid for online program construction.

Implementation on the IBM 1130 computer

This AMTRAN version was implemented on an IBM 1130 computer with 8K of core and a disc. A typewriter version is currently being tested, and a multiconsole version is under development.

Some of the goals for the implementation were high speed, fully dynamic storage allocation, powerful editing and checking capabilities, and a completely re-entrant structure for multiconsole use with short response time. The length of programs, the dimension of arrays, the ratio of program area to variable area, the number of defined variables and programs can be chosen with complete freedom as long as the available core storage, dependent upon implementation, is not exceeded.

A special monitor system, independent of IBM software, has been developed for a more efficient use of the disc to obtain short response times.

AMTRAN as a tool for pedagogic purposes

The interactive system AMTRAN is highly useful not only for research purposes but also as an educational tool. Lowering the level of difficulty in programming makes the computing facility available for students who are not basically interested in computer science but want to expand their understanding of mathematics or physics. Graphic display capabilities are very well suited for studying and demonstrating the behavior of functions. By interacting directly with the computer, the student also gets a better feeling for the kind of problems involved in programming a computer.

A multiconsole system eliminates keypunch problems, and there are none of the time delay, debugging, or control language problems usually found in batch mode.

COMPARISON WITH OTHER HIGH LEVEL LANGUAGES

A comparative study between AMTRAN and other high level languages has to be divided into two parts. Only language features can be compared with batch mode languages, whereas the whole AMTRAN- system can be taken into account for a comparison with other interactive systems.

Batch mode languages

Most likely, PL/1, ALGOL, or FORTRAN would be used to solve mathematical, technical or scientific problems in batch mode. A comparison with AMTRAN is not really feasible as the basic philosophy and design principles of batch mode languages are completely different from AMTRAN.

Since language development goes more and more in the direction of powerful general purpose languages, it becomes more and more difficult, time consuming, and cumbersome for the nonprogrammer to make the first step towards use of a computer. But even for the experienced user, the three languages mentioned above do not provide the convenience and facilities in programming that AMTRAN does. They need type and dimension declarations; the flexibility in changing types and dimensions at run time is lacking; and they do not have AMTRAN's array handling capabilities.

PL/1 with its default philosophy, its various types of storage allocation, and certain automatic array arithmetic features is close to AMTRAN's facilities and philosophy of programming convenience. On the other hand, it is inconvenient for the user to keep track of storage allocation problems in writing recursive or re-entrant programs or in using arrays with computed origin.

PL/1 is truly a general purpose programming language. It is designed for programming needs in science, engineering, data management, and business. AMTRAN, on the other hand, is a special purpose programming language for mathematical, scientific, and technical applications and has not been designed to compete in general with a language like PL/1. It is not intended to handle extensive data; therefore, it does not need powerful I/O-capabilities and sophisticated formatting facilities. But it can compete or even perform better within the limits of its special purpose.

Interactive systems

An interactive console system fills the gap between a desk top calculator and conventional batch mode

computer programming. On one hand, it has to give immediate answers to simple requests; on the other hand, it has to provide powerful programming capabilities.

A milestone in the development of interactive systems was the Culler-Fried-System, which strongly influenced the early AMTRAN development. Prof. Culler's system represents a highly powerful multi-console system. A disadvantage is that it does not stay close to the mathematical notation, and it is not simple and easy to learn.[7]

CPS is a conversational subset of PL/1. It has a calculator mode and a program mode and is a useful conversational system although it does not have AMTRAN's flexibility and power in array and function handling.

Iverson's language APL (A Programming Language) is a more formal approach to application programming. It is particularly useful for classical mathematical applications, and it has been implemented as a powerful interactive time-sharing system. The language has features such as array arithmetic, programming capabilities, and a large set of primitive operators including matrix handling operators. An extensive set of special symbols is used instead of keywords. Thus, a special typewriter is necessary. The proponents of APL claim that its source code is more efficient per statement than that of any other programming language. On the other hand, it is less readable. One has to learn special symbols instead of using mnemonics. For example, the quad □ in APL is less informative as an output operator than the TYPE in AMTRAN.

Major disadvantages are that APL does not follow classical mathematical notation, there is no hierarchy among operators, and the order of execution of statements is from right to left. This means the mathematician and scientific nonprogrammer must convert his formulas, written in normal textbook format, into the APL-notation, and the programmer experienced in any other language is even more confused. APL is a language which requires both care and training for simple applications.

CONCLUSIONS

AMTRAN, as described in this paper, can be implemented on a small computer. Such a small computing system is a serious alternative to a console of a commercial time-sharing system. The present developments on the hardware market—a decrease in the price of small computers—make the outlook for problem-solving systems like AMTRAN, particularly bright.

ACKNOWLEDGMENT

This work has been supported in part by National Science Foundation grant GJ-39.

BIBLIOGRAPHY

1 J REINFELDS L A FLENKER R N SEITZ P L CLEM
Proceedings of the Annual Meeting of the ACM p 469
Thompson Press New York 1966

2 G KRATKY H KOPETZ
The semantics of a mathematically oriented computer language
Proceedings of the ACM National Conference San Francisco August 1969

3 G KRATKY
The definition of AMTRAN
Computer Center AMTRAN Report University of Georgia 1969-4

4 B D FRIED
On the users point of view
Interactive Systems for Experimental Applied Mathematics M Klerer and J Reinfelds eds Academic Press New York 1968

5 P NAUR et al
Revised report on the algorithmic language ALGOL 60
Communications of the ACM Vol 6 pp 1-17 January 1963

6 N ESKELSON H KOPETZ
AMTRAN-manual for the IBM 130 computer
To be Published 1969

7 W D ORR
Conversational computers
John Wiley and Sons Inc 1968

Computer network development to achieve resource sharing

by LAWRENCE G. ROBERTS and BARRY D. WESSLER

Advanced Research Projects Agency
Washington, D.C.

INTRODUCTION

In this paper a computer network is defined to be a set of autonomous, independent computer systems, interconnected so as to permit interactive resource sharing between any pair of systems. An overview of the need for a computer network, the requirements of a computer communication system, a description of the properties of the communication system chosen, and the potential uses of such a network are described in this paper.

The goal of the computer network is for each computer to make every local resource available to any computer in the net in such a way that any program available to local users can be used remotely without degradation. That is, any program should be able to call on the resources of other computers much as it would call a subroutine. The resources which can be shared in this way include software and data, as well as hardware. Within a local community, time-sharing systems already permit the sharing of software resources. An effective network would eliminate the size and distance limitations on such communities. Currently, each computer center in the country is forced to recreate all of the software and data files it wishes to utilize. In many cases this involves complete reprogramming of software or reformatting the data files. This duplication is extremely costly and has led to considerable pressure for both very restrictive language standards and the use of identical hardware systems. With a successful network, the core problem of sharing resources would be severely reduced, thus eliminating the need for stifling language standards. The basic technology necessary to construct a resource sharing computer network has been available since the advent of time-sharing. For example, a time-sharing system makes all its resources available to a number of users at remote consoles. By splicing two systems together as remote users of each other and permitting user programs to interact with two consoles (the human user and the remote computer), the basic characteristics of a network connection are obtained. Such an experiment was made between the TX–2 computer at Lincoln Lab and the Q–32 computer at SDC in 1966 in order to test the philosophy.[1] Logically, such an interconnection is quite powerful and one can tap all the resources of the other system. Practically, however, the interconnection of pairs of computers with console grade communication service is virtually useless. First, the value of a network to a user is directly proportional to the number of other workers on the net who are creating potentially useful resources. A net involving only two systems is therefore far less valuable than one incorporating twenty systems. Second, the degradation in response caused by using telegraph or voice grade communication lines for network connections is significant enough to discourage most users. Third, the cost to fully interconnect computers nation-wide either with direct leased lines or dial-up facilities is prohibitive. All three problems are a direct result of the inadequacy of the available communication services.

DESIGN OF A NETWORK COMMUNICATIONS SERVICE

After the Lincoln-SDC network experiments, it was clear that a completely new communications service was required in order to make an effective, useful resource-sharing computer network. The communication pipelines offered by the carriers would probably have to be a component of that service but were clearly inadequate by themselves. What was needed was a message service where any computer could submit a message destined for another computer and be sure it would be delivered promptly and correctly. Each interactive

conversation or link between two computers would have messages flowing back and forth similar to the type of traffic between a user console and a computer. Message sizes of from one character to 1000 characters are characteristic of man-machine interactions and this should also be true for that network traffic where a man is the end consumer of the information being exchanged. Besides having a heavy bias toward short messages, network traffic will also be diverse. With twenty computers, each with dozens of time-shared users, there might be, at peak times, one or more conversations between all 190 pairs of computers.

Reliability

Communications systems, being designed to carry very redundant information for direct human consumption, have, for computers, unacceptably high downtime and an excessively high error rate. The line errors can easily be fixed through error detection and retransmission; however, this does require the use of some computation and storage at both ends of each communication line. To protect against total line failures, there should be at least two physically separate paths to route each message. Otherwise the service will appear to be far too unreliable to count on and users will continue to duplicate remote resources rather than access them through the net.

Responsiveness

In those cases where a user is making more or less direct use of a complete remote software system, the network must not cause the total round-trip delay to exceed the human short-term memory span of one to two seconds. Since the time-sharing systems probably introduce at least a one-second delay, the network's end-to-end delay should be less than ½ second. The network response should also be comparable, if possible, to using a remote display console over a private voice grade line where a 50 character line of text (400 bits) can be sent in 200 ms. Further, if interactive graphics are to be available, the network should be able to send a complete new display page requiring about 20 kilobits of information within a second and permit interrupts (10–100) to get through very quickly, hopefully within 30–90 ms. Where two programs are interacting without a human user being directly involved, the job will obviously get through sooner, the shorter the message delay. There is no clear critical point here, but if the communications system substantially slows up the job, the user will probably choose to duplicate the remote process or data at his site. For such cases, a reasonable measure by which to compare communications systems is the "effective bandwidth" (data block length for the job/end-to-end transmission delay).

Capacity

The capacity required is proportional to the number and variety of services available from the network. As the number of nodes increase, the traffic is expected to increase more than linearly, until new nodes merely duplicate available network resources. The number of nodes in the experimental network was chosen to: (1) involve as many computer researchers as possible to develop network protocol and operating procedures, (2) involve special facilities, such as the ILLIAC, to distribute its resources to a wider community, (3) involve as many disciplines of science as possible to measure the effect of the network on those disciplines, and (4) involve many different kinds of computers and systems to prove the generality of the techniques developed. The nodes of the network were generally limited to: (1) those centers for which the network would truly provide a cost benefit, (2) government-funded projects because of the use of special rate government-furnished communications, and (3) ARPA-funded projects where the problems of intercomputer accounting could be deferred until the network was in stable operation. The size of the experimental network was chosen to be approximately 20 nodes nation-wide. It was felt that this would be large and diverse enough to be a useful utility and to provide enough traffic to adequately test the network communication system.

For a 20 node network, the total traffic by mid-1971 at peak hours is estimated to be 200-800 KB (kilobits per second). This corresponds to an average outgoing traffic per node of 10–40 KB or an average of 0.5–2 KB traffic both ways between each pair of nodes. Traffic between individual node-pairs, however, will vary considerably, from zero to 10 KB. The total traffic per node will also vary widely, perhaps from 5–50 KB. Variations of these magnitudes will occur in both space and time and, unless the connumications system can reallocate capacity rapidly (seconds), the users will find either the delay or cost excessive. However, it is expected that the total capacity required for all 20 nodes will be fairly stable, smoothed out by having hundreds of active network users spread out across four time zones.

Cost

To be a useful utility, it was felt that communications costs for the network should be less than 25% of the

computing costs of the systems connected through the network. This is in contrast to the rising costs of remote access communications which often cost as much as the computing equipment.

If we examine why communications usually cost so much we find that it is not the communications channel per se, but our inefficient use of them, the switching costs, or the operations cost. To obtain a perspective on the price we commonly pay for communications let us evaluate a few methods. As an example, let us use a distance of 1400 miles since that is the average distance between pairs of nodes in the projected ARPA Network. A useful measure of communications cost is the cost to move one million bits of information, cents/megabit. In the table below this is computed for each media. It is assumed for leased equipment and data set rental that the usage is eight hours per working day.

TABLE 1—Cost per Megabit for Various Communication Media 1400 Mile Distance

Media		
Telegram	$3300.00	For 100 words at 30 bits/wd, daytime
Night Letter	565.00	For 100 words at 30 bits/wd, overnight delivery
Computer Console	374.00	18 baud avg. use[2], 300 baud DDD service line & data sets only
TELEX	204.00	50 baud teletype service
DDD (103A)	22.50	300 baud data sets, DDD daytime service
AUTODIN	8.20	2400 baud message service, full use during working hours
DDD (202)	3.45	2000 baud data sets
Letter	3.30	Airmail, 4 pages, 250 wds/pg, 30 bits/wd
W. U. Broadband	2.03	2400 baud service, full duplex
WATS	1.54	2000 baud, used 8 hrs/working day
Leased Line (201)	.57	2000 baud, commercial, full duplex
Data 50	.47	50 KB dial service, utilized full duplex
Leased Line (303)	.23	50 KB, commercial, full duplex
Mail DEC Tape	.20	2.5 megabit tape, airmail
Mail IBM Tape	.034	100 megabit tape, airmail

Special care has also been taken to minimize the cost of the multiplexor or switch. Previous store and forward systems like DoD's AUTODIN system, have had such complex, expensive switches that over 95% of the total communications service cost was for the switches. Other switch services adding to the system's cost, deemed superfluous in a computer network, were: long term message storage, multi-address messages and individual message accounting.

The final cost criteria was to minimize the communications software development cost required at each node site. If the network software could be generated centrally, not only would the cost be significantly reduced, but also the reliability would be significantly enhanced.

THE ARPA NETWORK

Three classes of communications systems were investigated as candidates for the ARPA Network: fully interconnected point to point leased lines, line switched (dial-up) service, and message switched (store and forward) service. For the kind of service required, it was decided and later verified that the message switched service provided the greater flexibility, higher effective bandwidth, and lower cost than the other two systems.

The standard message switched service uses a large central switch with all the nodes connected to the switch via communication lines; this configuration is generally referred to as a Star. Star systems perform satisfactorily for large blocks of traffic (greater than 100 kilobits per message), but the central switch saturates very quickly for small message sizes. This phenomenon adds significant delay to the delivery of the message. Also, a Star design has inherently poor reliability since a single line failure can isolate a node and the failure of the central switch is catastrophic.

An alternative to the Star, suggested by the Rand study "On Distributed Communications"[3], is a fully distributed message switched system. Such a system has a switch or store and forward center at every node in the network. Each node has a few transmission lines to other nodes; messages are therefore routed from node to node until reaching their destination. Each transmission line thereby multiplexes messages from a large number of source-destination pairs of nodes. The distributed store and forward system was chosen, after careful study, as the ARPA Network communications system. The properties of such a communication system are described below and compared with other systems.

A more complete description of the implementation, optimization, and initial use of the network can be found in a series of five papers, of which this is the first. The second paper by Heart, et al[4] describes the design, implementation and performance characteristics of the message switch. The third paper by Kleinrock[5] derives procedures for optimizing the capacity of the trans-

mission facility in order to minimize cost and average message delay. The fourth paper by Frank, et al[6] describes the procedure for finding optimized network topologies under various constraints. The last paper by Carr, et al[7] is concerned with the system software required to allow the network computers to talk to one another. This final paper describes a first attempt at intercomputer protocol, which is expected to grow and mature as we gain experience in computer networking.

Network properties

The switching centers use small general purpose computers called Interface Message Processors (IMPs) to route messages, to error check the transmission lines and to provide asynchronous digital interface to the main (HOST) computer. The IMPs are connected together via 50 Kbps data transmission facilities using common carrier (ATT) point to point leased lines. The topology of the network transmission lines was selected to minimize cost, maximize growth potential, and yet satisfy all the design criteria.

Reliability

The network specification requires that the delivered message error rates be matched with computer characteristics, and that the down-time of the communications system be extrememly small. Three steps have been taken to insure these reliability characteristics: (1) at least two transmission paths exist between any two modes, (2) a 24 bit cyclic check sum is provided for each 1000 bit block of data, and (3) the IMP is ruggedized against external environmental conditions and its operation is independent of any electromechanical devices (except fans). The down-time of the transmission facility is estimated at 10–12 hours per year (no figures are currently available from ATT). The duplication of paths should result in average down-time between any pair of nodes, due to transmission failure, of approximately 30 seconds per year. The cyclic check sum was chosen based on the performance characteristics of the transmission facility; it is designed to detect long burst errors. The code is used for error detection only, with retransmission on an error. This check reduces the undetected bit error rate to one in 10^{12} or about one undetected error per year in the entire network.

The ruggedized IMP is expected to have a mean time to failure of 10,000 hours; less than one failure per year. The elimination of mass storage devices from the IMP results in lower cost, less down-time, and greater throughput performance of the IMP, but implies no long term message storage and no message accounting by the IMP. If these functions are later needed they can be added by establishing a special node in the network. This node would accept accounting information from all the IMPs and also could be routed all the traffic destined for HOSTs which are down. We do not believe these functions are necessary, but the network design is capable of providing them.

Responsiveness

The target goal for responsiveness was .5 seconds transit time from any node to any other, for a 1000 bit (or less) block of information. The simulations of the network show the transit time of a 1 kilobit block of .1 seconds until the network begins to saturate. After saturation the transit time rises quickly because of excessive queuing delays. However, saturation will hopefully be avoided by the net acting to choke off the inputs for short periods of time, reducing the buffer queues while not significantly increasing the delay.

Capacity

The capacity of the network is the throughput rate at which saturation occurs. The saturation level is a function of the topology and capacity of the transmission lines, the traffic distribution between pairs of nodes (traffic matrix) and the average size of the blocks sent over the transmission lines. The analysis of capacity was performed by Network Analysis Corporation during the optimization of the network topology. As the analysis shows, the network has the ability to flexibly increase its capacity by adding additional transmission lines. The use of 108 and 230.4 KB communication services, where appropriate, considerably improves the cost-performance of the network.

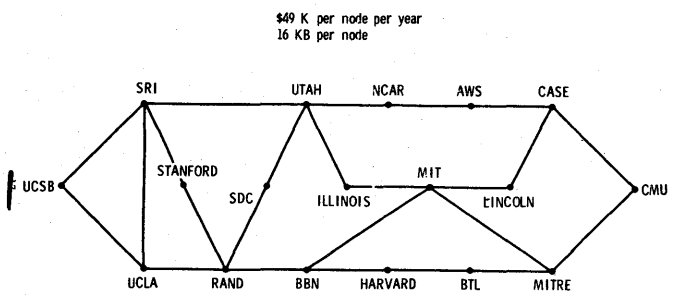


Figure 1—ARPA network initial topology

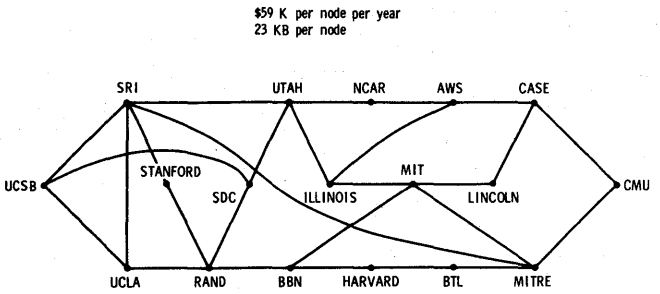


Figure 2—ARPA network expanded topology

Configuration

Initial configuration of the ARPA Network is currently planned as shown in Figure 1. The communications circuits for this network will cost $49K per node per year and the network can support an average traffic of 16 KB per node. If the traffic builds up, additional communication lines can be added to expand the capacity as required. For example, if 23 KB per node is desired, the network can be expanded to the configuration shown in Figure 2 for an increase of only $10K per node per year. Expansion can be continued on this basis until a capacity of about 60 KB per node is achieved, at which point the IMPs would tend to saturate.

COMPARISON WITH ALTERNATIVE NETWORK COMMUNICATIONS SYSTEMS DESIGNS

For the purpose of this comparison the capacity required was set at 500 baud to 1 KB per node-pair. A minimal buffer for error checking and retransmission at every node is included in the cost of the systems.

Two comparisons are made between the systems: the cost per megabit as a function of the delay and the effective bandwidth as a function of the block size of the data. Several other functions were plotted and compared; the two chosen were deemed the most informative. The latter graph is particularly informative in showing the effect of using the network for short, interactive message traffic.

The systems chosen for the comparison were fully interconnected 2.4 KB and 19 KB leased line systems, Data-50 the dial-up 50 KB service, DDD the standard 2 KB voice grade dial-up system, Star networks using 19 KB and 50 KB leased lines into a central switch, and the ARPA Network using 50 KB leased lines.

The graph in Figure 3 shows the cost per megabit versus delay. The rectangle outlines the variation caused by a block size variation of 1 to 10 Kilobits and

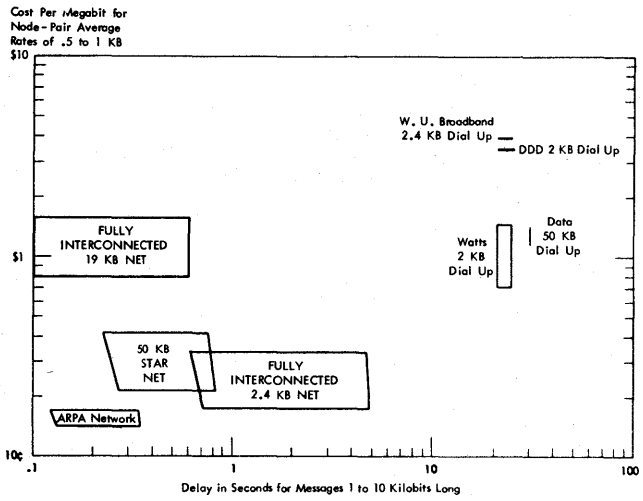


Figure 3—Cost vs delay for potential 20 node network designs

capacity requirement variation of 500 to 1000 baud. The dial-up systems were used in a way to minimize the line charges while keeping the delay as low as possible. The technique is to dial a system, then transmit the data accumulated during the dial-up (20 seconds for DDD, 30 seconds for Data-50). The dial-up systems are still very expensive and slow as compared with other alternatives. The costs of the ARPA Network are for optimally designed topologies. The 19 KB Star was eliminated because the system saturated just below 1 KB per node-pair which did not provide adequate growth potential though the cost was comparable to the ARPA Network. For the 50 KB Star network, the switch is assumed to be an average distance of 1300 miles from every node.

The graph in Figure 4 shows the effective bandwidth

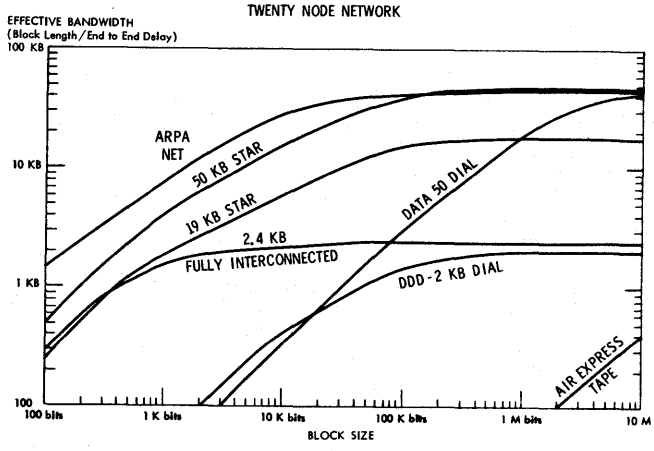


Figure 4—Effective Bandwidth vs block size

versus the block size of the data input to the network. The curves for the various systems are estimated for traffic rates of 500 to 1000 baud. The comparison shows the ARPA Net does very well at small block size where most of the traffic is expected.

NETWORK PLANS

Use of the Network is broken into two successive phases: (1) Initial Research and Experimental Use, and (2) External Research Community Use. These phases are closely related to our plans for Network implementation. The first phase, started in September 1969, involves the connection of 14 sites involved principally in computer research. These sites are current ARPA contractors who are working in the areas of Computer System Architecture, Information System Design, Information Handling, Computer Augmented Problem Solving, Intelligent Systems, as well as Computer Networks. This phase should be fully implemented by November 1970. The second phase involves the extension of the number of sites to about 20 to include ARPA-supported research disciplines.

Initial research and experimental use

During Phase One, the community of users will number approximately 2000 people. This community is involved primarily in computer science research and all have ARPA-funded on-going research. The major use they will make of the network is the sharing of software resources and the educational experience of using a wider variety of systems than previously possible. The software resources available to the Network include: advanced user programs such as MATHLAB at MIT, Theorem Provers at SRI, Natural Language Processors at BBN, etc., and new system software and languages such as LEAP, a graphic language at Lincoln Lab, LC^2, an interactive ALGOL system at Carnegie, etc.

Another major use of the Network will be for accessing the Network Information Center (NIC). The NIC is being established at SRI as the repository of information about all systems connected into the Network. The NIC will maintain, update and distribute hard copy information to all users. It will also provide file space and a system for accessing and updating (through the net) dynamic information about the systems, such as system modifications, new resources available, etc.

The final major use of the Net during Phase One is for measurement and experimentation on the Network itself. The primary sites involved in this are BBN, who has responsibility for system development and system maintenance, and UCLA, who has responsibility for the Net measurement and modeling. All the sites will also be involved in the generation of intercomputer protocol, the language the systems use to talk to one another.

External research community use

During the time period after November 1970, additional nodes will be installed to take advantage of the Network in three other ARPA-funded research disciplines: Behavioral Science, Climate Dynamics and Seismology. The use of the Network at these nodes will be oriented more toward the distribution and sharing of stored data, and in the latter two fields the use of the ILLIAC IV at the University of Illinois.

The data sharing between data management systems or data retrieval systems will begin an important phase in the use of the Network. The concept of distributed data bases and distributed access to the data is one of the most powerful and useful applications of the network for the general data processing community. As described above, if the Network is responsive in the human time frame, data bases can be stored and maintained at a remote location rather than duplicating them at each site the data is needed. Not only can the data be accessed as if the user were local, but also as a Network user he can write programs on his own machine to collect data from a number of locations for comparison, merging or further analysis.

Because of widespread use of the ILLIAC IV, it will undoubtably be the single most demanding node in the Network. Users will not only be sending requests for service but will also send very large quantities of input and output data, e.g., a 10^6 bit weather map, over the Net. Projected uses of the ILLIAC include weather and climate modeling, picture processing, linear programming, matrix manipulations, and extensive work in other areas of simulation and modeling.

In addition to the ILLIAC, the University of Illinois will also have a trillion bit mass store. An experiment is being planned to use 10% of the storage (100 billion bits) as archival storage for all the nodes on the Net. This kind of capability may help reduce the number of tape drives and/or data cells in the Network.

FUTURE

There are many applications of computers for which current communications technology is not adequate. One such application is the specialized customer service computer systems in existence or envisioned for the

future; these services provide the customer with information or computational capability. If no commercial computer network service is developed, the future may be as follows:

One can envision a corporate officer in the future having many different consoles in his office: one to the stock exchange to monitor his own company's and competitor's activities, one to the commodities market to monitor the demand for his product or raw materials, one to his own company's data management system to monitor inventory, sales, payroll, cash flow, etc., and one to a scientific computer used for modeling and simulation to help plan for the future. There are probably many people within that same organization who need some of the same services and potentially many other services. Also, though the data exists in digital form on other computers, it will probably have to be keypunched into the company's modeling and simulation system in order to perform analyses. The picture presented seems rather bleak, but is just a projection of the service systems which have been developed to date.

The organization providing the service has a hard time, too. In addition to collecting and maintaining the data, the service must have field offices to maintain the consoles and the communications multiplexors adding significantly to their cost. A large fraction of that cost is for communications and consoles, rather than the service itself. Thus, the services which can be justified are very limited.

Let us now paint another picture given a nationwide network for computer-to-computer communication. The service organization need only connect its computer into the net. It probably would not have any consoles other than for data input, maintenance, and system development. In fact, some of the service's data input may come from another service over the Net. Users could choose the service they desired based on reliability, cleanliness of data, and ease of use, rather than proximity or sole source.

Large companies would connect their computers into the net and contract with service organizations for the use of those services they desired. The executive would then have one console, connected to his company's machine. He would have one standard way of requesting the service he desires with a far greater number of services available to him.

For the small company, a master service organization might develop, similar to today's time-sharing service, to offer console service to people who cannot afford their own computer. The master service organization would be wholesalers of the services and might even be used by the large companies in order to avoid contracting with all the individual service organizations.

The kinds of services that will be available and the cost and ultimate capacity required for such service is difficult to predict. It is clear, however, that if the network philosophy is adopted and if it is made widely available through a common carrier, that the communications system will not be the limiting factor in the development of these services as it is now.

REFERENCES

1 T MARILL L ROBERTS
Toward a cooperative network of time-shared computers
AFIPS Conference Proceedings Nov 1966

2 P E JACKSON C D STUBBS
A study of multi-access computer communications
AFIPS Conference Proceedings Vol 34 p 491 1969

3 PAUL BARAN et al
On distributed communications
RAND Series Reports Aug 1964

4 F E HEART R E KAHN S M ORNSTEIN W R CROWTHER D C WALDEN
The interface Message Processor for the ARPA network
AFIPS Conference Proceedings May 1970

5 L KLEINROCK
Analytic and simulation methods in Computer network design
AFIPS Conference Proceedings May 1970

6 H FRANK IT FRISCH W CHOU
Topological considerations in the design of the ARPA computer network
AFIPS Conference Proceedings May 1970

7 S CARR S CROCKER V CERF
HOST–HOST Communication protocol in the ARPA network
AFIPS Conference Proceedings May 1970

The interface message processor for the ARPA computer network*

by F. E. HEART, R. E. KAHN, S. M. ORNSTEIN, W. R. CROWTHER and D. C. WALDEN

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

For many years, small groups of computers have been interconnected in various ways. Only recently, however, has the interaction of computers and communications become an important topic in its own right.** In 1968, after considerable preliminary investigation and discussion, the Advanced Research Projects Agency of the Department of Defense (ARPA) embarked on the implementation of a new kind of nationwide computer interconnection known as the ARPA Network. This network will initially interconnect many dissimilar computers at ten ARPA-supported research centers with 50-kilobit common-carrier circuits. The network may be extended to include many other locations and circuits of higher bandwidth.

The primary goal of the ARPA project is to permit persons and programs at one research center to access data and use interactively programs that exist and run in other computers of the network. This goal may represent a major step down the path taken by computer time-sharing, in the sense that the computer resources of the various research centers are thus pooled and directly accessible to the entire community of network participants.

Study of the technology and tariffs of available communications facilities showed that use of conventional *line switching* facilities would be economically and technically inefficient. The traditional method of routing information through the common-carrier switched network establishes a dedicated path for each conversation. With present technology, the time required for this task is on the order of seconds. For voice communication, that overhead time is negligible, but in the case of many short transmissions, such as may occur between computers, that time is excessive. Therefore, ARPA decided to build a new kind of digital communication system employing wideband leased lines and *message switching*, wherein a path is not established in advance and each message carries an address. In this domain the project portends a possible major change in the character of data communication services in the United States.

In a nationwide computer network, economic considerations also mitigate against a wideband leased line configuration that is topologically fully connected. In a non-fully connected network, messages must normally traverse several network nodes in going from source to destination. The ARPA Network is designed on this principle and, at each node, a copy of the message is stored until it is safely received at the following node. The network is thus a store and forward system and as such must deal with problems of routing, buffering, synchronization, error control, reliability, and other related issues. To insulate the computer centers from these problems, and to insulate the network from the problems of the computer centers, ARPA decided to place identical small processors at each network node, to interconnect these small processors with leased common-carrier circuits to form a *subnet*, and to connect each research computer center into the net via the local small processor. In this arrangement the research computer centers are called *Hosts* and the small processors are called *Interface Message Processors*, or *IMPs*. (See Figure 1.) This approach divides the genesis of the ARPA Network into two parts: (1) design and implementation of the IMP subnet, and (2) design and implementation of protocols and techniques for the sensible utilization of the network by the Hosts.

Implementation of the subnet involves two major

* This work was sponsored by the Advanced Research Projects Agency under Contract No. DAHC 15-69-C-0179.

** A bibliography of relevant references is included at the end of this paper; a more extensive list may be found in Cuadra, 1968.

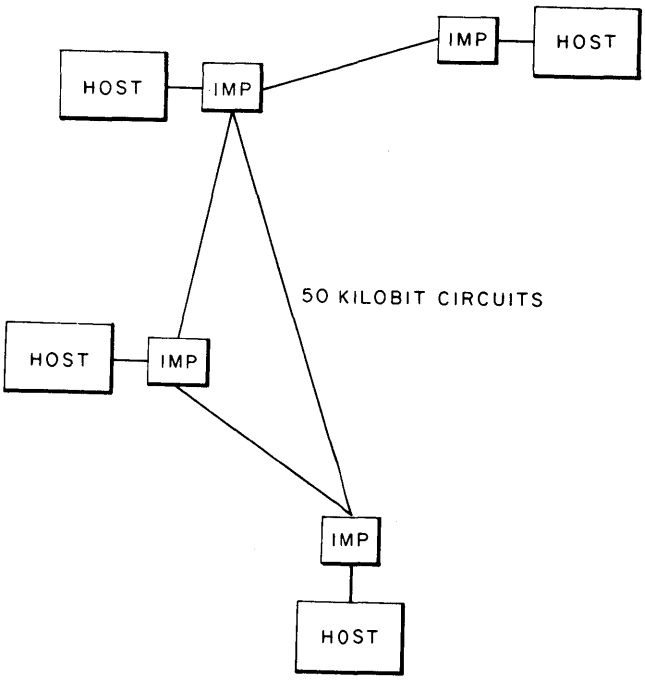


Figure 1—Hosts and IMPs

technical activities: providing 50-kilobit common-carrier circuits and the associated modems; and providing IMPs, along with software and interfaces to modems and Host computers. For reasons of economic and political convenience, ARPA obtained common-carrier circuits directly through government purchasing channels; AT&T (Long Lines) is the central coordinator, although the General Telephone Company is participating at some sites and other common carriers may eventually become involved. In January 1969, Bolt Beranek and Newman Inc. (BBN) began work on the design and implementation of IMPs; a four-node test network was scheduled for completion by the end of 1969 and plans were formulated to include a total of ten sites by mid-1970. This paper discusses the design of the subnet and describes the hardware, the software, and the predicted performance of the IMP. The issues of Host-to-Host protocol and network utilization are barely touched upon; these problems are currently being considered by the participating Hosts and may be expected to be a subject of technical interest for many years to come.

At this time, in late 1969, the test network has become an operating reality. IMPs have already been installed at four sites, and implementation of IMPs for six additional sites is proceeding. The common carriers have installed 50-kilobit leased service connecting the first four sites and are preparing to install circuits at six additional sites.

The design of the network allows for the connection of additional Host sites. A map of a projected eleven-node network is shown in Figure 2. The connections between the first four sites are indicated by solid lines. Dotted lines indicate planned connections.

NETWORK DESIGN

The design of the network is discussed in two parts. The first part concerns the relations between the Hosts and the subnet, and the second part concerns the design of the subnet itself.

Host-subnet considerations

The basic notion of a subnet leads directly to a series of questions about the relationship between the Hosts and the subnet: What tasks shall be performed by each? What constraints shall each place on the other? What dependence shall the subnet have on the Hosts? In considering these questions, we were guided by the following principles: (1) The subnet should function as a *communications system* whose essential task is to transfer bits reliably from a source location to a specified destination. Bit transmission should be sufficiently reliable and error free to obviate the need for special precautions (such as storage for retransmission) on the part of the Hosts; (2) The average transit time through the subnet should be under a half second to provide for convenient interactive use of remote computers; (3) The subnet operation should be completely autonomous. Since the subnet must function as a store and forward system, an IMP must not be dependent upon its local Host. The IMP must

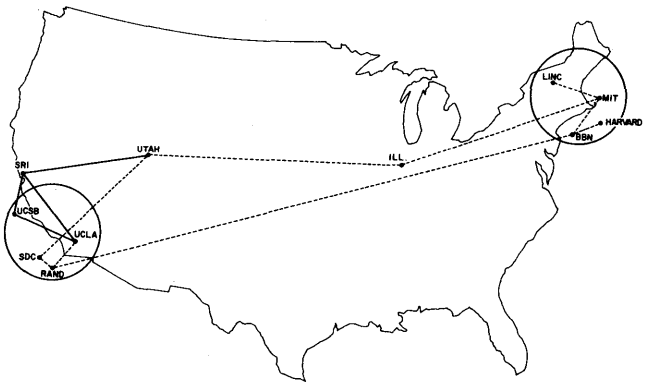


Figure 2—Network map

continue to operate whether the Host is functioning properly or not and must not depend upon a Host for buffer storage or other logical assistance such as program reloading. The Host computer must not in any way be able to change the logical characteristics of the subnet; this restriction avoids the mischievous or inadvertent modification of the communication system by an individual Host user; (4) Establishment of Host-to-Host protocol and the enormous problem of planning to communicate between different computers should be an issue separated from the subnet design.

Messages, links, and RFNMs

In principle, a single transmission from one Host to another may range from a few bits, as with a single teletype character, up to arbitrarily many bits, as in a very long file. Because of buffering limitations in the subnet, an upper limit was placed on the size of an individual Host transmission; 8095 bits was chosen for the maximum transmission size. This Host unit of transmission is called a *message*. The subnet does not impose any pattern restrictions on messages; binary text may be transmitted. Messages may be of variable length; thus, a source Host must indicate the end of a message to the subnet.

A major hazard in a message switched network is congestion, which can arise either due to system failures or to peak traffic flow. Congestion typically occurs when a destination IMP becomes flooded with incoming messages for its Host. If the flow of messages to this destination is not regulated, the congestion will back up into the network, affecting other IMPs and degrading or even completely clogging the communication service. To solve this problem we developed a quenching scheme that limits the flow of messages to a given destination when congestion begins to occur or, more generally, when messages are simply not getting through.

The subnet transmits messages over unidirectional logical paths between Hosts known as *links*. (A link is a conceptual path that has no physical reality; the term merely identifies a message sequence.) The subnet accepts only one message at a time on a given link. Ensuing messages on that link will be blocked from entering the subnet until the source IMP learns that the previous message has arrived at the destination Host. When a link becomes unblocked, the subnet notifies the source Host by sending it a special control message known as *Ready for Next Message* (or RFNM), which identifies the newly unblocked link. The source Host may utilize its connection into the subnet to transmit messages over other links, while waiting to send messages on the blocked links. Up to 63 separate outgoing links may exist at any Host site. When giving the subnet a message, the Host specifies the destination Host and a link number in the first 32 bits of the message (known as the *leader*). The IMPs then attend to route selection, delivery, and notification of receipt. This use of links and RFNMs also provides for IMP-to-Host delivery of sequences of messages in proper order. Because the subnet allows only one message at a time on a given link, Hosts never receive messages out of sequence.

Host-IMP interfacing

Each IMP will initially service a single Host. However, we have made provision (both in the hardware and software) for the IMP to service up to four Hosts, with a corresponding reduction in the number of permitted phone line connections. Connecting an IMP to a wide variety of different Hosts requires a hardware interface, some part of which must be custom tailored to each Host. We decided, therefore, to partition the interface such that a standard portion would be built into the IMP, and would be identical for all Hosts, while a special portion of the interface would be unique to each Host. The interface is designed to allow messages to flow in both directions at once. A bit serial interface was designed partly because it required fewer lines for electrical interfacing and was, therefore, less expensive, and partly to accommodate conveniently the variety of word lengths in the different Host computers. The bit rate requirement on the Host line is sufficiently low that parallel transfers are not necessary.

The Host interface operates asynchronously, each data bit being passed across the interface via a *Ready For Next Bit/There's Your Bit* handshake procedure. This technique permits the bit rate to adjust to the rate of the slower member of the pair and allows necessary interruptions, when words must be stored into or retrieved from memory. The IMP introduces between bits a (manually) adjustable delay that limits the maximum data rate; at present, this delay is set to 10 μsec. Any delay introduced by the Host in the handshake procedure further slows the rate.

System failure

Considerable attention has been given to the possible effects on a Host of system failures in the subnet. Minor system failures (e.g., temporary line failures) will appear to the Hosts only in the form of reduced rate of service. Catastrophic failures may, however, result in the loss of messages or even in the loss of

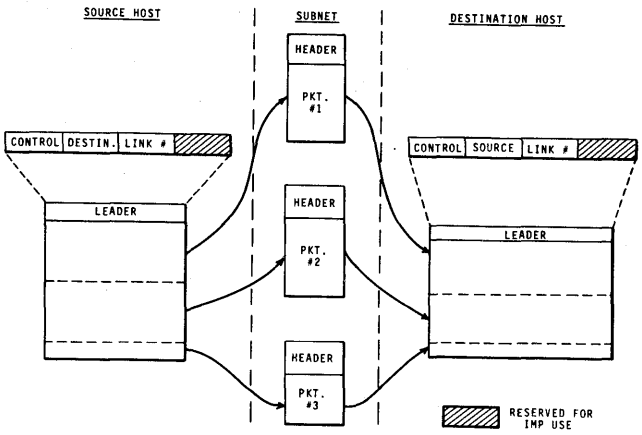


Figure 3—Messages and packets

subnet communication. IMPs inform a Host of all relevant system failures. Additionally, should a Host computer go down, the information is propagated throughout the subnet to all IMPs so they may notify their local Host if it attempts to send a message to that Host.

Specific subnet design

The overriding consideration that guided the subnet design was reliability. Each IMP must operate unattended and reliably over long periods with minimal down time for maintenance and repair. We were convinced that it was important for each IMP in the subnet to operate autonomously, not only independently of Hosts, but insofar as possible from other IMPs as well; any dependency between one IMP and another would merely broaden the area jeopardized by one IMP's failure. The need for reliability and autonomy bears directly upon the form of subnet communication. This section describes the process of message communication within the subnet.

Message handling

Hosts communicate with each other via a sequence of messages. An IMP takes in a message from its Host computer in segments, forms these segments into *packets* (whose maximum size is approximately 1000 bits), and ships the packets separately into the network. The destination IMP reassembles the packets and delivers them in sequence to the receiving Host, who obtains them as a single unit. This segmentation of a message during transmission is completely invisible to the Host computers. Figures 3, 4, and 5 illustrate aspects of message handling.

The transmitting Host attaches an identifying leader to the beginning of each message. The IMP forms a *header* by adding further information for network use and attaches this header to each packet of the message.

Each packet is individually routed from IMP-to-IMP through the network toward the destination. At each IMP along the way, the transmitting hardware generates initial and terminal framing characters and parity check digits that are shipped with the packet and are used for error detection by the receiving hardware of the next IMP.

Errors in transmission can affect a packet by destroying the framing and/or by modifying the data content. If the framing is disturbed in any way, the packet either will not be recognized or will be rejected by the receiver. In addition, the check digits provide protection against errors that affect only the data. The check digits can detect all patterns of four or fewer errors occurring within a packet, and any single error burst of a length less than twenty-four bits. An overwhelming majority of all other possible errors (all but about one in 2^{24}) are also detected. Thus, the mean time between undetected errors in the subnet should be on the order of years.

As a packet moves through the subnet, each IMP stores the packet until a positive acknowledgment is returned from the succeeding IMP. This acknowledgment indicates that the message was received without error and was accepted. Once an IMP has accepted a packet and returned a positive acknowledgment, it holds onto that packet tenaciously until it in turn receives an acknowledgment from the succeeding IMP. Under no circumstances (except for Host or IMP malfunction) will an IMP discard a packet after it has generated a positive acknowledgment. However, an IMP is always free to refuse a packet by simply not returning a positive acknowledgment. It may do this for any of several reasons: the packet may have

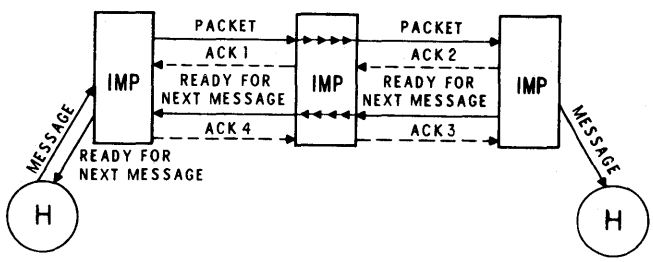


Figure 4—RFNMs and acknowledgments

been received in error, the IMP may be busy, the IMP buffer storage may be temporarily full, etc.

At the transmitting IMP, such discard of a packet is readily detected by the absence of a returned acknowledgment within a reasonable time interval (e.g., 100 msec). Such packets are retransmitted, perhaps along a different route. Acknowledgments themselves are not acknowledged, although they are error checked in the usual fashion. Loss of an acknowledgment results in the eventual retransmission of the packet; the destination IMP sorts out the resulting duplication by using a message number and a packet number in the header.

The packets of a message arrive at the destination IMP, possibly out of order, where they are reassembled. The header is then stripped off each packet and a leader, identifying the source Host and the link, followed by the reassembled message is then delivered to the destination Host as a single unit. See Figure 3.

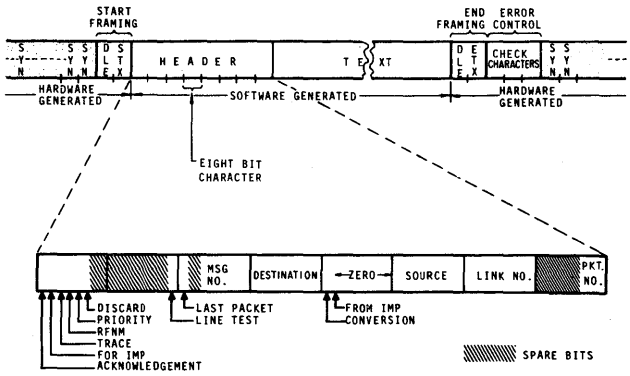


Figure 5—Format of packet on phone line

Routing algorithm

The routing algorithm directs each packet to its destination along a path for which the total estimated transit time is smallest. This path is not determined in advance. Instead, each IMP individually decides onto which of its output lines to transmit a packet addressed to another destination. This selection is made by a fast and simple table lookup procedure. For each possible destination, an entry in the table designates the appropriate next leg. These entries reflect line or IMP trouble, traffic congestion, and current subnet connectivity. This routing table is updated every halfsecond as follows:

Each IMP estimates the delay it expects a packet to encounter in reaching every possible destination over each of its output lines. It selects the minimum delay estimate for each destination and periodically (about twice a second) passes these estimates to its immediate neighbors. Each IMP then constructs its own routing table by combining its neighbors' estimates with its own estimates of the delay to that neighbor. The estimated delay to each neighbor is based upon both queue lengths and the recent performance of the connecting communication circuit. For each destination, the table is then made to specify that selected output line for which the sum of the estimated delay to the neighbor plus the neighbor's delay to the destination is smallest.

The routing table is consistently and dynamically updated to adjust for changing conditions in the network. The system is adaptive to the ups and downs of lines, IMPs, and congestion; *it does not require the IMP to know the topology of the network.* In particular, an IMP need not even know the identity of its immediate neighbors. Thus, the leased circuits could be reconfigured to a new topology without requiring any changes to the IMPs.

Subnet failures

The network is designed to be largely invulnerable to circuit or IMP failure as well as to outages for maintenance. Special status and test messages are employed to help cope with various failures. In the absence of regular packets for transmission over a line, the IMP program transmits special *hello* packets at half-second intervals. The acknowledgment for a hello packet is an *I heard you* packet.

A *dead line* is defined by the sustained absence (approximately 2.5 seconds) on that line of either received regular packets or acknowledgments; no regular packets will be routed onto a dead line, and any packets awaiting transmission will be rerouted. Routing tables in the network are adjusted automatically to reflect the loss. We require acknowledgment of thirty consecutive *hello* packets (an event which consumes at least 15 seconds), before a dead line is defined to be alive once again.

A dead line may reflect trouble either in the communication facilities or in the neighboring IMP itself. Normal line errors caused by dropouts, impulse noise, or other conditions should not result in a dead line, because such errors typically last only a few milliseconds, and only occasionally as long as a few tenths of a second. Therefore, we expect that a line will be defined as dead only when serious trouble conditions occur. If dead lines eliminate all routes between two IMPs, the IMPs are said to be *disconnected* and each

of these IMPs will discard messages destined for the other. Disconnected IMPs cannot be rapidly detected from the delay estimates that arrive from neighboring IMPs. Consequently, additional information is transmitted between neighboring IMPs to help detect this condition. Each IMP transmits to its neighbors the length of the shortest existing path (i.e., number of IMPs) from itself to each destination. To the smallest such received number per destination, the IMP adds one. This incremented number is the length of the shortest path from that IMP to the destination. If the length ever exceeds the number of network nodes, the destination IMP is assumed to be unreachable and therefore disconnected.

Messages intended for dead Hosts (which are not the same as dead IMPs) cannot be delivered; therefore, these messages require special handling to avoid indefinite circulation in the network and spurious arrival at a later time. Such messages are purged from the network either at the source IMP or at the destination IMP. Dead Host information is regularly transmitted with the routing information. A Host computer is notified about another dead Host only when attempting to send a message to that Host.

An IMP may detect a major failure in one of three ways: (1) A packet expected for reassembly of a multiple packet message does not arrive. If a message is not fully reassembled in 15 minutes, the system presumes a failure. The message is discarded by the destination IMP and both the source IMP and the source Host are notified via a special RFNM. (2) The Host does not take a message from its IMP. If the Host has not taken a message after 15 minutes, the system presumes that it will never take the message. Therefore, as in the previous case, the message is discarded and a special RFNM is returned to the source Host. (3) A link is never unblocked. If a link remains blocked for longer than 20 minutes, the system again presumes a failure; the link is then unblocked and an error message is sent to the source Host. (This last time interval is slightly longer than the others so that the failure mechanisms for the first two situations will have a chance to operate and unblock the link.)

Reliability and recovery procedures

For higher system reliability, special attention was placed on intrinsic reliability, hardware test capabilities, hardware/software failure recovery techniques, and proper administrative mechanisms for failure management.

To improve intrinsic reliability, we decided to ruggedize the IMP hardware, thus incurring an approximately ten percent hardware cost penalty. For ease in maintenance, debugging, program revision, and analysis of performance, all IMPs are as similar as possible; the operational program and the hardware are nearly identical in all IMPs.

To improve hardware test capabilities, we built special *crosspatching* features into the IMP's interface hardware; these features allow program-controlled connection of output lines to corresponding input lines. These crosspatching features have been invaluable in testing IMPs before and during field installation, and they should continue to be very useful when troubles occur in the operating network. These hardware test features are employed by a special hardware test program and may also be employed by the operational program when a line difficulty occurs.

The IMP includes a 512-word block of protected memory that secures special recovery programs. An IMP can recover from an IMP failure in two ways: (1) In the event of power failure, a power-fail interrupt permits the IMP to reach a clean stop before the program is destroyed. When power returns, a special automatic restart feature turns the IMP back on and restarts the program. (We considered several possibilities for handling the packets found in an IMP during a power failure and concluded that no plan to salvage the packets was both practical and foolproof. For example, we cannot know whether the packet in transmission at the time of failure successfully left the machine before the power failed. Therefore, we decided simply to discard all the packets and restart the program.) (2) The second recovery mechanism is a "watchdog timer", which transfers control to protected memory whenever the program neglects this timer for about one minute. In the event of such transfer, the program in unprotected memory is presumed to be destroyed (either through a hardware transient or a software failure). The program in protected memory sends a reload request down a phone line *selected at random.* The neighboring IMP responds by sending a copy of its whole program back on the phone line. A normal IMP would discard this message because it is too long, but the recovering IMP can use it to reload its program.

Everything unique to a particular IMP must thus reside in its protected memory. Only one register (containing the IMP number) currently differs from IMP-to-IMP. The process of reloading, which requires a few seconds, can be tried repeatedly until successful; however, if after several minutes the program has not resumed operation, a later phase of the watchdog timer shuts off all power to the IMP.

In addition to providing recovery mechanisms for both network and IMP failures, we have incorporated

into the subnet a *control center* that monitors network status and handles trouble reports. The control center, located at a network node, initiates and follows up any corrective actions necessary for proper subnet functioning. Furthermore, this center controls and schedules any modifications to the subnet.

Introspection

Because the network is experimental in nature, considerable effort has been allocated to developing tools whereby the network can supply measures of its own performance. The operational IMP program is capable of taking statistics on its own performance on a regular basis; this function may be turned on and off remotely. The various kinds of resulting statistics, which are sent via the network to a selected Host for analysis, include "snapshots", ten-second summaries, and packet arrival times. Snapshots are summaries of the internal status of queue lengths and routing information. A synchronization procedure allows these snapshots, which are taken every half second, to occur at roughly the same time in all network IMPs; a Host receiving such snapshot messages could presumably build up an instantaneous picture of overall network status. Ten-second summaries include such IMP-generated statistics as the number of processed messages of each kind, the number of retransmissions, the traffic to and from the local Host, and so forth; this statistical data is sent to a selected Host every ten seconds. In addition, a record of actual packet arrival times on modem lines allows for the modeling of line traffic. (As part of its research activity, the group at UCLA is acting as a network measurement center; thus, statistics for analysis will normally be routed to the UCLA Host.)

Perhaps the most powerful capability for network introspection is *tracing*. Any Host message sent into the network may have a "trace bit" set in the leader. Whenever it processes a packet from such a message, the IMP keeps special records of what happens to that packet—e.g., how long the packet is on various queues, when it comes in and leaves, etc. Each IMP that handles the traced packet generates special trace report messages that are sent to a specified Host; thus, a complete analysis of what has happened to that message can be made. When used in an orderly way, this tracing facility will aid in understanding at a very detailed level the behavior of routing algorithms and the behavior of the network under changing load conditions.

Flexibility

Flexibility for modifications in IMP usage has been provided by several built-in arrangements: (1) provision within the existing cabinet for an additional 4K core bank; (2) modularity of the hardware interfaces; (3) provision for operation with data circuits of widely different rates; (4) a program organization involving many nearly self-contained subprograms; and (5) provision for Host-unique subprograms in the IMP program structure.

This last aspect of flexibility presents a somewhat controversial design choice. There are many advantages to keeping all IMP software nearly identical. Because of the experimental nature of the network, however, we do not yet know whether this luxury of identical programs will be an optimal arrangement. Several potential applications of "Host-unique" IMP software have been considered—e.g., using ASCII conversion routines in each IMP to establish a "Network ASCII" and possibly to simplify the protocol problems of each Host. As of now, the operational IMP program includes a *structure* that permits unique software plug-in packages at each Host site, but no plug-ins have yet been constructed.

THE HARDWARE

We selected a Honeywell DDP-516 for the IMP processor because we wanted a machine that could easily handle currently anticipated maximum traffic and that had already been proven in the field. We considered only economic machines with fast cycle times and good instruction sets. Furthermore, we needed a machine with a particularly good I/O capability and that was available in a ruggedized version. The geographical proximity of the supplier to BBN was also a consideration.

The basic machine has a 16-bit word length and a 0.96-μsec memory cycle. The IMP version is packaged in a single cabinet, and includes a 12K memory, a set of 16 multiplexed channels (which implement a 4-cycle data break), a set of 16 priority interrupts, a 100-μsec clock, and a set of programmable status lights. Also packaged within this cabinet are special modular interfaces for connecting the IMP to phone line modems and to Host computers; these interfaces use the same kind of 1 MHz and 5 MHz DTL packs from which the main machine is constructed. In addition, a number of features that have been incorporated make the IMP somewhat resilient to a variety of failures.

Teletypes and high-speed paper tape readers which are attached to the IMPs are used only for mainte-

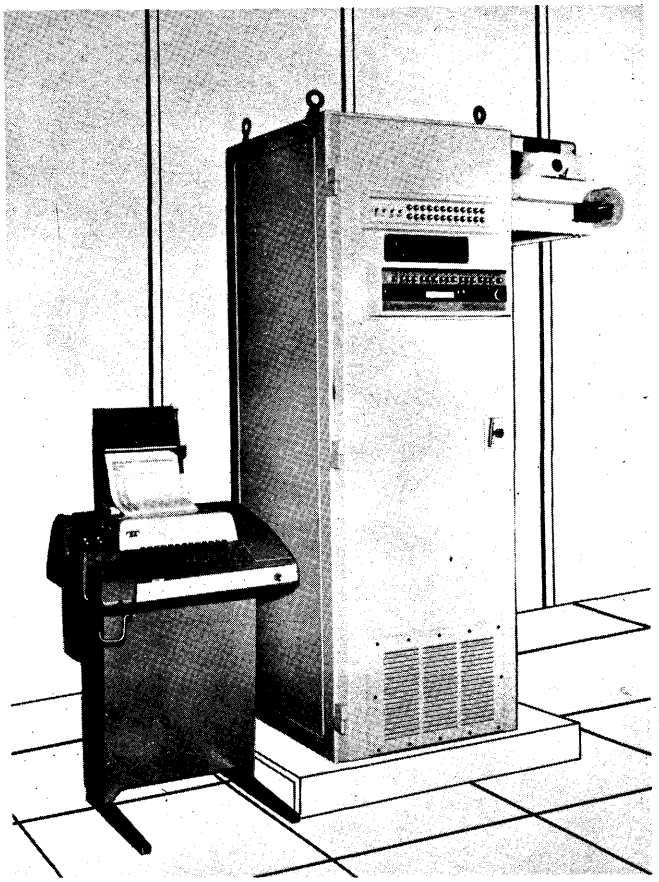


Figure 6—The IMP

nance, debugging, and system modification; in normal operation, the IMP runs without any moving parts except fans. Within the cabinet, space has been reserved for an additional 4K memory. Figure 6 is a picture of an IMP, and Figure 7 shows its configuration.

Ruggedization of computer hardware for use in friendly environments is somewhat unusual; however, we felt that the considerable difficulty that IMP failures can cause the network justified this step. Although the ruggedized unit is not fully "qualified" to MIL specs, it does have greater resistance to temperature variance, mechanical shock and vibration, radio frequency interference, and power line noise. We are confident that this ruggedization will increase the mean time to failure.

Modular Host and modem interfaces allow an IMP to be individually configured for each network node. The modularity, however, does not take the form of pluggable units and, except for the possibility of adding interfaces into reserved frame space, reconfiguration is impractical. Various configurations allow for up to two Hosts and five modems, three Hosts and four modems, etc. Each modem interface requires approximately one-fourth the amount of logic used in the C.P.U. The Host interface is somewhat smaller (about one-sixth of the C.P.U.).

Interfaces to the Host and to the modems have certain common characteristics. Both are full duplex, both may be crosspatched under program control to test their operation, and both function in the same general manner. To send a packet, the IMP program sets up memory pointers to the packet and then activates the interface via a programmable control pulse. The interface takes successive words from the memory using its assigned output data channel and transmits them bit-serially (to the Host or to the modem). When the memory buffer has thus been emptied, the interface notifies the program via an interrupt that the job has been completed. To receive information, the program first sets pointers to the allocated space in the memory into which the information is to flow. Using a control pulse it then readies the interface to receive. When information starts to arrive (here again bit-serially), it is assembled into 16-bit words and stored into the IMP memory. When either the allocated memory space is full or the end of the data train is detected, the interface notifies the program via an interrupt.

The modem interfaces deal with the phone lines in terms of 8-bit characters; the interfaces idle by sending and receiving a sync pattern that keeps them in character sync. Bit sync is maintained by the modems themselves, which provide both transmit and receive clocking signals to the interfaces. When the program initiates

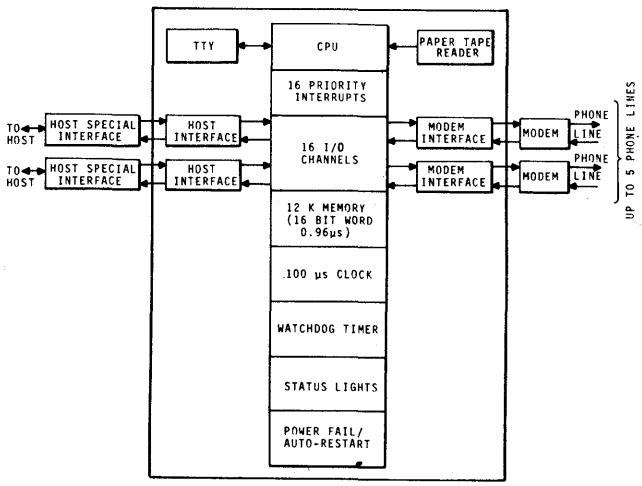


Figure 7—IMP configuration

transmission, the hardware first transmits a pair of initial framing characters (DLE, STX). Next, the text of the packet is taken word by word from the memory and shifted serially onto the phone line. At the end of the data, the hardware generates a pair of terminal framing characters (DLE, ETX) and shifts them onto the phone line. After the terminal framing characters, the hardware generates and transmits 24 check bits. Finally, the interface returns to idle (sync) mode.

The hardware doubles any DLE characters within the binary data train (that is, transmits them twice), thereby permitting the receiving interface hardware to distinguish them from the terminal framing characters and to remove the duplicate. Transmitted packets are of a known maximum size; therefore, any overflow of input buffer length is evidence of erroneous transmission. Format errors in the framing also register as errors. Check bits are computed from the received data and compared with the received check bits to detect errors in the text. Any of these errors set a flag and cause a program interrupt. Before processing a packet, the program checks the error flag to determine whether the packet was received correctly.

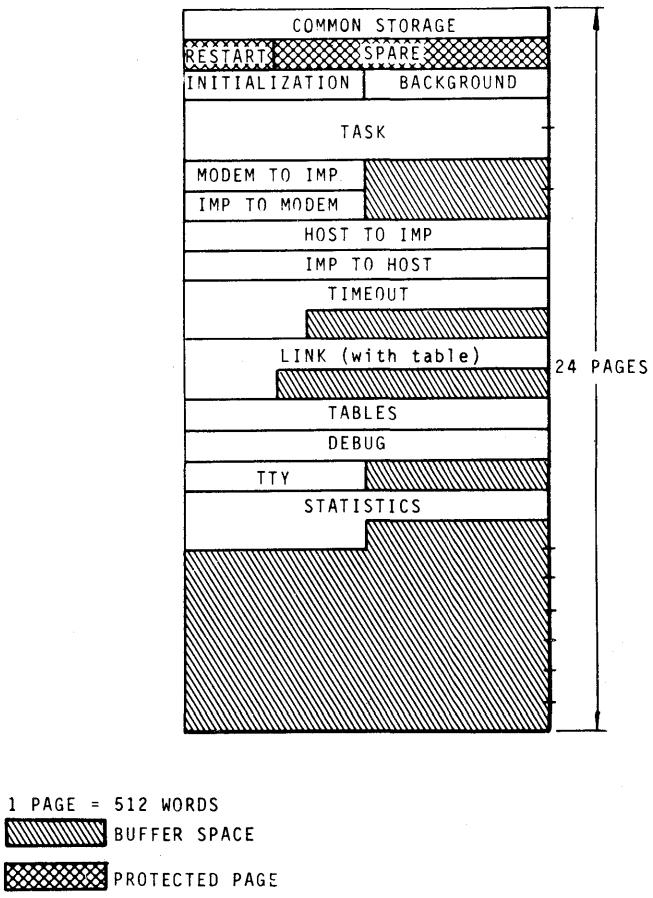


Figure 8—Map of core storage

IMP SOFTWARE

Implementation of the IMPs required the development of a sophisticated operational computer program and the development of several auxiliary programs for hardware tests, program construction, and debugging. This section discusses in detail the design of the operational program and briefly describes the auxiliary software.

Operational program

The principal function of the operational program is the processing of packets. This processing includes segmentation of Host messages into packets for routing and transmission, building of headers, receiving, routing and transmitting of store and forward packets, retransmitting of unacknowledged packets, reassembling received packets into messages for transmission to the Host, and generating of RFNMs and acknowledgments. The program also monitors network status, gathers statistics, and performs on-line testing. This real-time program is an efficient, interrupt-driven, involute machine language program that occupies about 6000 words of memory. It was designed, constructed, and debugged over a period of about a year by three programmers.

The entire program is composed of twelve functionally distinct pieces; each piece occupies no more than one or two pages of core (512 words per page). These programs communicate primarily through common registers that reside in page zero of the machine and that are directly addressable from all pages of memory. A map of core storage is shown in Figure 8. Seven of the twelve programs are directly involved in the flow of packets through the IMP: the *task* program performs the major portion of the packet processing, including the reassembly of Host messages; the *modem* programs (IMP-to-Modem and Modem-to-IMP) handle interrupts and resetting of buffers for the modem channels; the *Host* programs (IMP-to-Host and Host-to-IMP) handle interrupts and resetting of buffers for the Host channels, build packet headers during input, and construct RFNMs that are returned to the source Host during output; the *time-out* program maintains a software clock, times out unacknowledged packets for retransmission, and attends to infrequent events; the *link* program assigns and verifies message numbers and keeps track of links. A background loop

TABLE I—Program Data Structures

5000 WORDS—MESSAGE BUFFER STORAGE
120 WORDS—QUEUE POINTERS
300 WORDS—TRACE BLOCKS
100 WORDS—REASSEMBLY BLOCKS
150 WORDS—ROUTING TABLES
400 WORDS—LINK TABLES
300 WORDS—STATISTICS TABLES

contains the remaining five programs and deals with initialization, debugging, testing, statistics gathering and tracing. After a brief description of data structures, we will discuss packet processing in some detail.

Buffer allocation, queues, and tables

The major system data structures (see Table I) consist of buffers and tables. The buffer storage space is partitioned into about 70 fixed length buffers, each of which is used for storing a single packet. An unused buffer is chained onto a free buffer list and is removed from this list when it is needed to store an incoming packet. A packet, once stored in a buffer, is never moved. After a packet has been successfully passed along to its Host or to another IMP, its buffer is returned to the free list. The buffer space is partitioned in such a way that each process (store and forward, traffic, Host traffic, etc.) is always guaranteed some buffers. For the sake of program speed and simplicity, no attempt is made to retrieve the space wasted by partially filled buffers.

In handling store and forward traffic, all processing is on a per packet basis. Further, although traffic to and from Hosts is composed of *messages*, the IMP rapidly converts to dealing with packets; the Host transmits a message as a single unit but the IMP takes it one buffer at a time. As each buffer is filled, the program selects another buffer for input until the entire message has been provided for. These successive buffers will, in general, be scattered throughout the memory. An equivalent inverse process occurs on output to the Host after all packets of the message have arrived at the destination IMP. No attempt is ever made to collect the packets of a message into a contiguous portion of the memory.

Buffers currently in use are either dedicated to an incoming or an outgoing packet, chained on a queue awaiting processing by the program, or being processed. Occasionally, a buffer may be simultaneously found on two queues; this situation can occur when a packet is waiting on one queue to be forwarded and on another to be acknowledged.

There are four principal types of queues:

Task: Packets received on Host channels are placed on the Host task queue. All received acknowledgments, dead Host and routing information, *I heard you* and *hello* packets are placed on the system task queue; all other packets from the modems are placed on the modem task queue. The program services the system task queue first, then the Host task queue, and finally the modem task queue.

Output: A separate output queue is constructed for each modem channel and each Host channel. Each modem output queue is subdivided into an acknowledgment queue, a priority queue, a RFNM queue, and a regular message queue, which are serviced in that order. Each Host output queue is subdivided into a control message queue, a priority queue, and a regular message queue, which are also serviced in the indicated order.

Sent: A separate queue for each modem channel contains packets that have already been transmitted on that line but for which no acknowledgment has yet been received.

Reassembly: The reassembly queue contains those packets that are being reassembled into messages for the Host.

Tables in core are allocated for the storage of queue pointers, for trace blocks, for reassembly information, for statistics, and for links. Most noteworthy of these is the link table, which is used at the source IMP for assignment of message numbers and for blocking and unblocking links, and at the destination IMP to verify message numbers for sequence control.

Packet flow and program structure

Figure 9 is a schematic drawing of packet processing; the processing programs are described below.

The *Host-to-IMP* routine (H → I) handles messages being transmitted from the local site. The routine uses the leader to construct a header that is prefixed to each packet of the message. It also creates a link for the message if necessary, blocks the link, puts the packets of the message on the Host task queue for further processing by the task routine, and triggers the programmable task interrupt. The routine then acquires a free buffer and sets up a new input. The routine tests a hardware trouble indicator, verifies the message format, and checks whether or not the destination is dead, the link table is full, or the link blocked. The routine is serially reentrant and services all Hosts connected to the IMP.

The *Modem-to-IMP* routine (M → I) handles inputs from the modems. This routine consists of several identical routines, one for each modem channel. (Such duplication is useful to obtain higher speed.) This routine sets up an input buffer (normally obtained from the free list), places the received packet on the appropriate task queue, and triggers the programmable task interrupt. Should no free buffers be available for input, the buffer at the head of the modem task queue is preempted. If the modem task queue is also empty, the received packet is discarded by setting up its buffer for input. However, a sufficient number of free buffers are specifically reserved to assure that received acknowledgments, routing packets, and the like are rarely discarded.

The *task routine* uses the header information to direct packets to their proper destination. The task routine is driven by the task interrupt, which is set whenever a packet is put on a task queue. The task routine routes packets from the Host task queue onto an output queue determined from the routing algorithm.

For each packet on the modem task queue, the task routine first determines whether sufficient buffer space is available. If the IMP has a shortage of store and forward buffers, the buffers on the modem task queue are simply returned to the free list without further processing. Normally, however, an acknowledgment packet is constructed and put near the front of the appropriate modem output queue. The destination of the packet is then inspected. If the packet is not for the local site, the routing algorithm selects a modem output queue for the packet. If a packet for the local site is a RFNM, the corresponding link is unblocked and the RFNM is put on a queue to the Host. If the packet is not a RFNM, it is joined with others of the same message on the reassembly queue. Whenever a message is completely reassembled, the packets of the message are put on an output queue to the Host for processing by the IMP-to-Host routine.

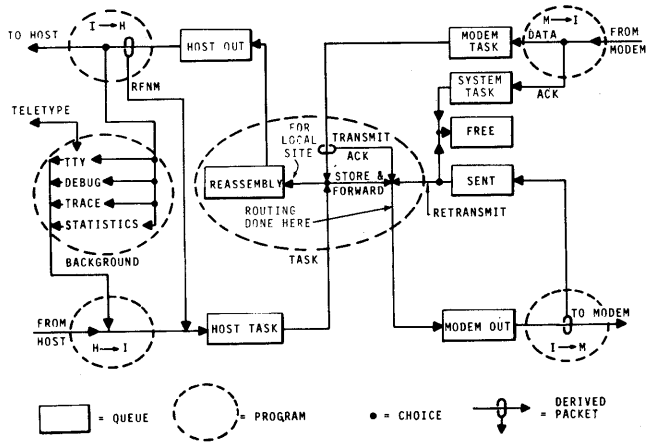


Figure 9—Internal packet flow

In processing the system task queue, the task routine returns to the free list those buffers from the sent queue that have been referenced by acknowledgments. Any packets skipped over by an acknowledgment are designated for retransmission. Routing, *I heard you*, and *hello* packets are processed in a straightforward fashion.

The *IMP-to-Modem* routine (I → M) transmits successive packets from the Modem output queue. After completing the output, this routine places any packet requiring acknowledgment on the sent queue.

The *IMP-to-Host* routine (I → H) sets up successive outputs of packets on the Host output queues and constructs a RFNM for each non-control message delivered to a Host. RFNM packets are returned to the system via the Host task queue.

The *time-out* routine is started every 25.6 msec (called the time-out period) by a clock interrupt. The routine has three sections: the fast time-out routine, which "wakes up" any Host or modem interrupt routine that has languished (for example, when the Host input routine could not immediately start a new input because of a shortage in buffer space); the middle time-out routine, which retransmits any packets that have been too long on a modem sent queue; and the slow time-out routine, which marks lines as alive or dead, updates the routing tables and does long term garbage collection of queues and other data structures. (For example, it protects the system from the cumulative effect of such failures as a lost packet of a multiple packet message, where buffers are tied up in message reassembly.) It also deletes links automatically after 15 seconds of disuse, after 20 minutes of blocking, or when an IMP goes down.

These three routines are executed in the following pattern:

FFFF FFFF FFFF FFFF FFFF FFFF . . .

M M M M M

S

and, although they run off a common interrupt, are constructed to allow faster routines to interrupt slower ones should a slower routine not complete execution before the next time-out period.

The *link* routine enters, examines, and deletes entries from the link table. A table containing a separate message number entry for many links to every possible Host would be prohibitively large. Therefore, the table contains entries only for each of 63 total out-

going links at any Host site. Hashing is used to speed accessing of this table, but the link program is still quite costly; it uses about ten percent of both speed and space in a conceptually trivial task.

Initialization and background loop

The IMP program starts in an initialization section that builds the initial data structures, prepares for inputs from modem and Host channels, and resets all program switches to their nominal state. The program then falls into the background loop, which is an endlessly repeated series of low-priority subroutines that are interrupted to handle normal traffic.

The programs in the IMP background loop perform a variety of functions: TTY is used to handle the IMP Teletype traffic; DEBUG, to inspect or change IMP core memory; TRACE, to transmit collected information about traced packets; STATISTICS, to take and transmit network and IMP statistics; PARAMETER-CHANGE, to alter the values of selected IMP parameters; and DISCARD, to throw away packets. Selected Hosts and IMPs, particularly the Network Measurement Center and the Network Control Center, will find it necessary or useful to communicate with one or more of these background loop programs. So that these programs may send and receive messages from the network, they are treated as "fake Hosts". Rather than duplicating portions of the large IMP-to-Host and Host-to-IMP routines, the background loop programs are treated as if they were Hosts, and they can thereby utilize existing programs. The "For IMP" bit or the "From IMP" bit in the leader indicates that a given message is for or from a fake Host program in the IMP. Almost all of the background loop is devoted to running these programs.

The TTY program assembles characters from the Teletype into network messages and decodes network messages into characters for the Teletype; TTY's normal message destination is the DEBUG program at its own IMP; however, TTY can be made to communicate with any other IMP Teletype, any other IMP DEBUG program or any Host program with compatible format.

The DEBUG program permits the operational program to be inspected and changed. Although its normal message source is the TTY program at its own IMP, DEBUG will respond to a message of the correct format from any source. This program is normally inhibited from changing the operational IMP program; local operator intervention is required to activate the program's full power.

The STATISTICS program collects measurements about network operation and periodically transmits them to the Network Measurement Center. This program sends but does not receive messages. STATISTICS has a mechanism for collecting measurements over 10-second intervals and for taking half-second snapshots of IMP queue lengths and routing tables. It can also generate artificial traffic to load the network. When turned on, STATISTICS uses 10 to 20 percent of the machine capacity and generates a noticeable amount of phone line traffic.

Other programs in the background loop drive local status lights and operate the parameter change routine. A thirty-two word parameter table controls the operation of the TRACE and STATISTICS programs and includes spares for expansion; the PARAMETER-CHANGE program accepts messages that change these parameters.

Control organization

It is characteristic of the IMP system that many of the main programs are entered both as subroutine calls from other programs and as interrupt calls from the hardware. The resulting control structure is shown in Figure 10. The programs are arranged in a priority order; control passes upward in the chain whenever a hardware interrupt occurs or the current program decides that the time has come to run a higher priority program, and control passes downward only when the higher priority programs are finished. No program may execute either itself or a lower priority program; however, a program may freely execute a higher priority program. This rule is similar to the usual rules concerning priority interrupt routines.

In one important case, however, control must pass from a higher priority program to a lower priority program—namely, from the several input routines to the TASK routine. For this special case, the computer hardware was modified to include a low-priority hardware interrupt that *can be set by the program.* When this interrupt has been honored (i.e., when all other interrupts have been serviced), the TASK routine is executed. Thus, control is directed where needed without violating the priority rules.

Some routines must occasionally wait for long intervals of time, for example, when the Host-to-IMP routine must wait for a link to unblock. Stopping the whole system would be intolerable; therefore, should the need arise, such a routine is dismissed, and the TIMEOUT routine will later transfer control to the waiting routine.

The control structure and the partition of responsibility among various programs achieve the following

timing goals:

1. No program stops or delays the system while waiting for an event.
2. The program gracefully adjusts to the situation where the machine becomes compute-bound.
3. The Modem-to-IMP routine can deliver its current packet to the TASK routine before the next packet arrives and can always prepare for successive packet inputs on each line. This timing is critical because a slight delay here might require retransmission of the entire packet. To achieve this result, separate routines (one per phone line) interrupt each other freely after new buffers have been set up.
4. The program will almost always deliver packets waiting to be sent as fast as they can be accepted by the phone line.
5. Necessary periodic processes (in the time-out routine) are always permitted to run, and do not interfere with input-output processes.

Support software

Designing a real-time program for a small computer with many high rate I/O channels is a specialized kind of software problem. The operational program requires not only unusual techniques but also extra software tools; often the importance of such extra tools is not recognized. Further, even when these issues are recognized, the effort needed to construct such tools may be seriously underestimated. The development of the IMP system required the following kinds of supporting software:

1. Programs to test the hardware.
2. Tools to help debug the system.
3. A Host simulator.
4. An efficient assembly process.

So far, three hardware test programs have been developed. The first and largest is a complete program for testing all the special hardware features in the IMP. This program permits running any or all of the modem interfaces in a crosspatched mode; it even permits operating together *several* IMPs in a test mode. The second hardware test program runs a detailed phone line test that provides statistics on phone line errors. The final program simulates the modem interface check register whose complex behavior is otherwise difficult to predict.

The software debugging tools exist in two forms. Initially we designed a simple stand-alone debugging program with the capability to do little more than examine and change individual core registers from the

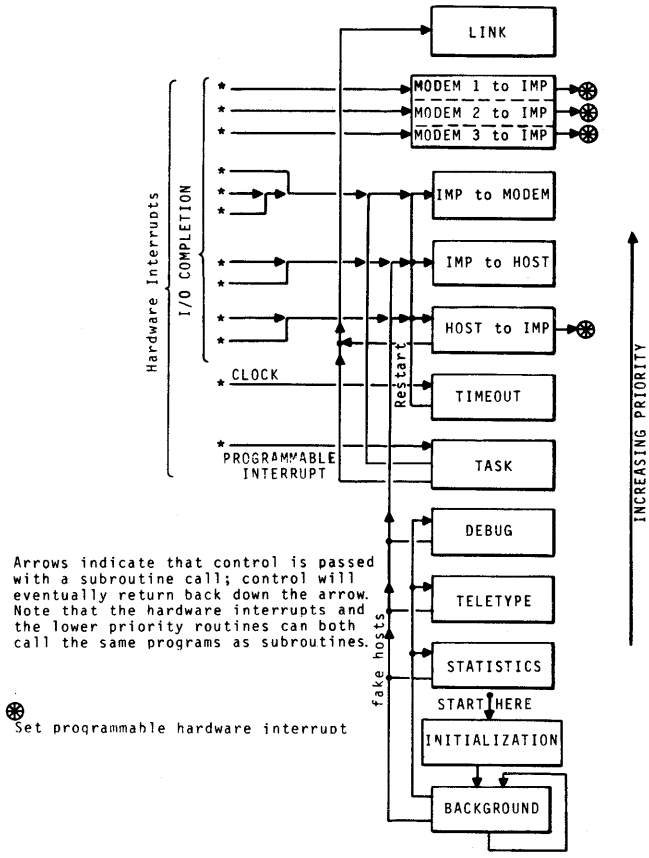


Figure 10—Program control structure

console Teletype. Subsequently, we embedded a version of the stand-alone debugging program into the operational program. This operational debugging program not only provides debugging assistance at a single location but also may be used in *network testing* and *network debugging.*

The initial implementation of the IMP software took place without connecting to a true Host. To permit checkout of the Host-related portions of the operational program, we built a "Host Simulator" that takes input from the console Teletype and feeds the Host routines exactly as though the input had originated in a real Host. Similarly, output messages for a destination Host are received by the simulator and typed out on the console Teletype.

Without recourse to expensive additional peripherals, the assembly facilities on the DDP-516 are inadequate for a large program. (For example, a listing of the IMP program would require approximately 20 hours of Teletype output.) We therefore used other locally available facilities to assist in the assembly process. Specifically, we used a PDP-1 text editor to compose and edit the programs, assembled on the

TABLE II—Transit Times And Message Rates

	Minimum	*Maximum*
SINGLE WORD MESSAGE		
Transit Time	5 msec	50 msec
Round-trip	10 msec	100 msec
Max. Message Rate/Link	100/sec	10/sec
SINGLE FULL PACKET MESSAGE		
Transit Time	45 msec	140 msec
Round-trip	50 msec	190 msec
Max. Message Rate/Link	20/sec	5/sec
8-PACKET MESSAGE		
Transit Time	265 msec	360 msec
Round-trip	195 msec	320 msec
Max. Message Rate/Link	5/sec	3/sec

DDP-516, and listed the program on the SDS 940 line printer. Use of this assembly process required minor modification of existing PDP-1 and SDS 940 support software

PROJECTED IMP PERFORMANCE

At this writing, the subnet has not yet been subjected to realistic load conditions; consequently, very little experimental data is available. However, we have made some estimates of projected performance of the IMP program and we describe these estimates below.

Host traffic and message delays

In the subnet, the Host-to-Host transit time and the round-trip time (for RFNM receipt) depend upon routing and message length. Since only one message at a time may be present on a given link, the reciprocal of the round-trip delay is the maximum message rate on a link. The primary factors affecting subnet delays are:

- Propagation delay: Electrical propagation time in the Bell system is estimated to be about 10 μsec per mile. Cross country propagation delay is therefore about 30 msec.
- Modem transmission delay: Because bits enter and leave an IMP at a predetermined modem bit rate, a packet requires a modem transmission time proportional to its length (20 μsec per bit on a 50-kilobit line).
- Queueing delay: Time spent waiting in the IMP for transmission of previous packets on a queue. Such waiting may occur either at an intermediate IMP or in connection with terminal IMP transmissions into the destination Host.
- IMP processing delay: The time required for the IMP program to process a packet is about 0.35 msec for a store-and-forward packet.

Because the queueing delay depends heavily upon the detailed traffic load in the network, an estimate of queueing delay will not be available until we gain considerable experience with network operation. In Table II, we show an estimate of the one-way and round-trip transit times and the corresponding maximum message rate per link, assuming the negligible queueing delay of a lightly loaded net. In this table, "minimum" delay represents a short hop between two nearby IMPs, and "maximum" delay represents a cross-country path involving five IMPs. In all cases the delays are well within the desired half-second goal.

In a lightly-loaded network with a mixture of nearby and distant destinations, an example of heavy Host traffic into its IMP might be that of 20 links carrying ten single-word messages per second and four more links, each carrying one eight-packet message per second.

Computational load

In general, a line fully loaded with short packets will require more computation than a line with all long packets; therefore the IMP can handle more lines in the latter case. In Figure 11, we show a curve of the computational utilization of the IMP as a function of message length for fully-loaded communication lines. For example, a 50-kilobit line fully loaded in both directions with one-word messages requires slightly over 13 percent of the available IMP time. Since a line will typically carry a variety of different length packets, and each line will be less than fully loaded, the computational load per line will actually be much less.

Throughput is defined to be the maximum number of Host data bits that may traverse an IMP each second. The actual number of bits entering the IMP per second is somewhat larger than the throughput because of such overhead as headers, RFNMs, and acknowledgments. The number of bits on the lines are still larger because of additional line overhead such as framing and error control characters. (Each packet on the phone line contains seventeen characters of

overhead, nine of which are removed before the packet enters an IMP.)

The computational limit on the IMP throughput is approximately 700,000 bits per second. Figure 12 shows maximum throughput as a function of message length. The difference between the throughput curve and the line traffic curve represents overhead.

DISCUSSION

In this section we state some of our conclusions about the design and implementation of the ARPA Network and comment on possible future directions.

We are convinced that use of an IMP-like device is a more sensible way to design networks than is use of direct Host-to-Host connection. First, for the subnet to serve a store-and-forward role, its functions must be independent of Host computers, which may often be down for extended periods. Second, the IMP program is very complex and is highly tailored to the I/O structure of the DDP-516; building such complex functions into special I/O units of each computer that might need network connection is probably economically inadvisable. Third, because of the desirability of having several Host computers at a given site connect to the network, it is both more convenient and more economic to employ IMPs than to provide all the network functions in each of the Host computers. The whole notion of a network node serving a multiplexing function for complexes of local Hosts and terminals lends further support to this conclusion. Finally, because we were led to a design having *some* inter-IMP dependence; we found it advantageous to have *identical* units at each node, rather than computers of different manufacture.

Considering the multiplexing issue directly, it now seems clear that individual network nodes will be connected to a wide variety of computer and terminal complexes. Even the initial ten-node ARPA Network

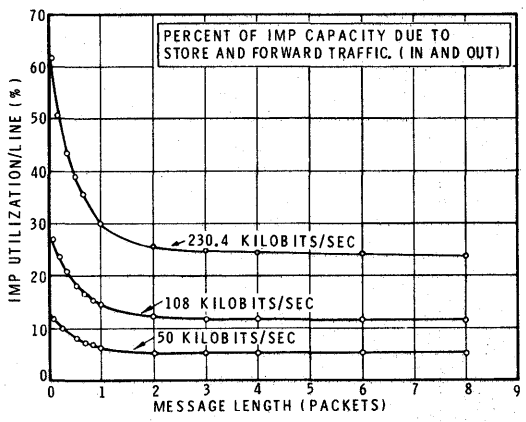


Figure 11—IMP utilization

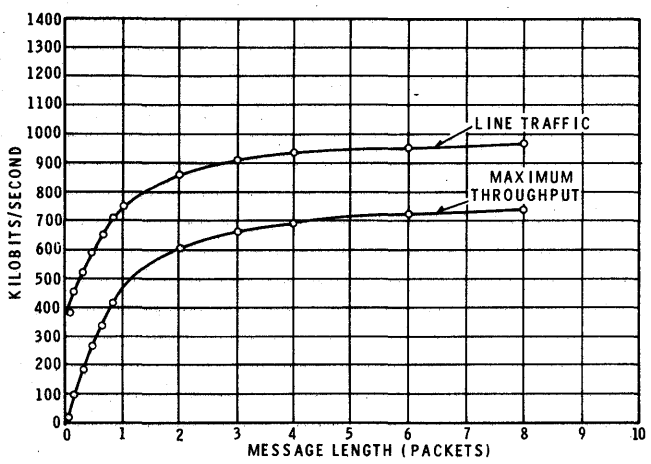


Figure 12—IMP throughput

includes one Host organization that has chosen to submultiplex several computers via a single Host connection to the IMP. We are now studying variants of the IMP design that address this multiplexing issue, and we also expect to cooperate with other groups (such as at the National Physical Laboratory in England) that are studying such multiplexing techniques.

The increasing interest in computer networks will bring with it an expanding interaction between computers and communication circuits. From the outset, we viewed the ARPA Network as a systems engineering problem, including the portion of the system supplied by the common carriers. Although we found the carriers to be properly concerned about circuit performance (the basic circuit performance to date has been quite satisfactory), we found it difficult to work with the carriers *cooperatively* on the technical details, packaging, and implementation of the communication circuit terminal equipment; as a result, the present physical installations of circuit terminal equipment are at best inelegant and inconvenient. In the longer run, for reasons of economy, performance, and reliability, circuit terminal equipment probably should be integrated more closely with computer input/output equipment. If the carriers are unable to participate conveniently in such integrations, we would expect further growth of a competing circuit terminal equipment industry, and more prevalent common carrier provision of bare circuits.

Another aspect of network growth and development is the requirement to connect different rate communication circuits to IMP-like devices as a function of the particular application. In our own IMP design, although there are limitations on total throughput,

the IMP can be connected to carrier circuits of any bit rate up to about 250 kilobits; similarly, the interface to a Host computer can operate over a wide range of bit rates. We feel that this flexibility is very important because the economics of carrier offerings, as well as the user requirements, are subject to surprisingly rapid change; even within the time period of the present implementation, we have experienced such changes.

At this point, we would like to discuss certain aspects of the implementation effort. This project required the design, development, and installation of a very complex device in a rather short time scale. The difficulty in producing a complex system is highly dependent upon the number of people who are simultaneously involved. Small groups can achieve complex optimizations of timing, storage, and hardware/software interaction, whereas larger groups can seldom achieve such optimizations on a reasonable time scale. We chose to operate with a very small group of highly talented people. For example, all software, including software tools for assembly, editing, debugging, and equipment testing as well as the main operational program, involved effort by no more than four people at any time. Since so many computer system projects involve much larger groups, we feel it is worth calling attention to this approach.

Turning to the future, we plan to work with the ARPA Network project along several technical directions: (1) the experimental operation of the network and any modifications required to tune its performance; (2) experimental operation of the network with higher bandwidth circuits, e.g., 230.4 kilobits; (3) a review of IMP variants that might perform multiplexing functions; (4) consideration of techniques for designing more economical and/or more powerful IMPs; and (5) participation with the Host organizations in the very sizeable problem of developing techniques and protocols for the effective *utilization* of the network.

On a more global level, we anticipate an explosive growth of message switched computer networks, not just for the interactive pooling of resources, but for the simple conveniences and economies to be obtained for many classes of digital data communication. We believe that the capabilities inherent in the design of even the present subnet have broad application to other data communication problems of government and private industry.

ACKNOWLEDGMENTS

The ARPA Network has in large measure been the conception of one man, Dr. L. G. Roberts of the Advanced Research Projects Agency; we gratefully acknowledge his guidance and encouragement. Researchers at many other institutions deserve credit for early interactions with ARPA concerning basic network design; in particular we would like to acknowledge the insight about IMPs provided by W. A. Clark.

At BBN, many persons contributed to the IMP project. We acknowledge the contributions of H. K. Rising, who participated in the subnet design and acted as associate project manager during various phases of the project; B. P. Cosell, who participated significantly in the software implementation; W. B. Barker and M. J. Thrope, who participated significantly in the hardware implementation; and T. Thatch, J. H. Geisman, and R. C. Satterfield, who assisted with various implementation aspects of the project. We also acknowledge the helpful encouragement of J. I. Elkind and D. G. Bobrow.

Finally, we wish to acknowledge the hardware implementation contribution of the Computer Control Division of Honeywell, where many individuals worked cooperatively with us despite the sometimes abrasive pressures of a difficult schedule.

REFERENCES

1 P BARAN
On distributed communication networks
IEEE Transactions on Communication Systems Vol CS-12 March 1964

2 P BARAN S BOEHM P SMITH
On distributed communications
Series of 11 reports Rand Corporation Santa Monica California 1964

3 B W BOEHM R L MOBLEY
Adaptive routing techniques for distributed communication systems
Rand Corporation Memorandum RM-4781-PR 1966

4 *Initial design for interface message processors for the ARPA computer network*
Bolt Beranek and Newman Inc Report No 1763 1969

5 *Specifications for the interconnection of a Host and an IMP*
Bolt Beranek and Newman Inc Report No 1822 1969

6 G W BROWN J G MILLER T A KEENAN
EDUNET report of the summer study on information networks conducted by the interuniversity communications council
John Wiley and Sons New York 1967

7 S CARR S CROCKER V CERF
HOST-HOST communication protocol in the ARPA network
Proceedings of AFIPS SJCC 1970 In this issue

8 C A CUADRA
Annual review of information science and technology
Interscience Vol 3 Chapters 7 and 10 1968

9 D W DAVIES K A BARTLETT
R A SCANTLEBURY P T WILKINSON
A digital communication network for computers giving rapid response at remote terminals
ACM Symposium on Operating System Principles 1967

10 D W DAVIES
The principles of a data communication network for computers and remote peripherals
Proceedings of IFIP Hardware Paper D11 1968
11 D W DAVIES
Communications networks to serve rapid-response computers
Proceedings of IFIP Edinburgh 1968
12 *EIN software catalogue*
EDUCOM 100 Charles River Park Boston (Regularly updated)
13 R R EVERETT C A ZRAKET H D BENINGTON
Sage–a data processing system for air defense
Proceedings of EJCC 1957
14 *Policies and regulatory procedures relating to computer and communication services*
Notice of Inquiry Docket No 16979 Washington D C 1966 Federal Communications Commission
15 L R FORD JR D R FULKERSON
Flows in networks
Princeton University Press 1962
16 H FRANK I T FRISCH W CHOU
Topological considerations in the design of the ARPA computer network
Proceedings of AFIPS SJCC 1970 In this issue
17 R T JAMES
The evolution of wideband services
IEEE International Convention Record Part I Wire and Data Communication 1966
18 S J KAPLAN
The advancing communication technology and computer communication systems
Proceedings of AFIPS SJCC Vol 32 1968
19 L KLEINROCK
Communications nets-stochastic message flow and delay
McGraw-Hill Book Co Inc New York 1964
20 L KLEINROCK
Models for computer networks
Proceedings of International Communications Conference June 1969
21 L KLEINROCK
Optimization of computer networks for various channel cost functions
Proceedings of AFIPS SJCC 1970 In this issue
22 T MARILL
Cooperative networks of time-shared computers
Computer Corporation of America Preliminary Study 1966 Also Private Report Lincoln Laboratory MIT Cambridge Massachusetts 1966
23 T MARILL L G ROBERTS
Toward a cooperative network of time-shared computers
Proceedings of AFIPS FJCC 1966
24 *Biomedical communications network-technical development Plan*
National Library of Medicine June 1968
25 *Networks of computers symposium NOC-68*
Proceedings of Invitational Workshop Ft Meade Maryland National Security Agency September 1969
26 *Networks of computers symposium NOC-69*
Proceedings of Invitational Workshop Ft Meade Maryland (in press) National Security Agency
27 M N PERRY W R PLUGGE
American Airlines 'Sabre' electronic reservations system
Proceedings of AFIPS WJCC 1961
28 L G ROBERTS
Multiple computer networks and intercomputer communication
ACM Symposium on Operating System Principles 1967
29 L G ROBERTS
Access control and file directories in computer networks
IEEE International Convention March 1968
30 L G ROBERTS
Resource sharing computer networks
IEEE International Conference March 1969
31 L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
Proceedings of AFIPS SJCC 1970 In this issue
32 R A SCANTLEBURY P T WILKINSON K A BARTLETT
The design of a message switching centre for a digital communication network
D26 Proceedings of IFIP Hardware Edinburgh 1968
33 K STEIGLITZ P WEINER D J KLEITMAN
The design of minimum cost survivable networks
IEEE Transactions on Circuit Theory Vol CT-16 November 1969
34 R SUNG J B WOODFORD
Study of communication links for the biomedical communication network
Aerospace Report No ATR-69 (7130-06)-1 1969
35 W TEITELMAN R E KAHN
A network simulation and display program
Proceedings of 3rd Annual Princeton Conference on Information Sciences and Systems March 1969

Analytic and simulation methods in computer network design*

by LEONARD KLEINROCK

University of California
Los Angeles, California

INTRODUCTION

The Seventies are here and so are computer networks! The time sharing industry dominated the Sixties and it appears that computer networks will play a similar role in the Seventies. The need has now arisen for many of these time-shared systems to share each others' resources by coupling them together over a communication network thereby creating a computer network. The mini-computer will serve an important role here as the sophisticated terminal as well as, perhaps, the message switching computer in our networks.

It is fair to say that the computer industry (as is true of most other large industries in their early development) has been guilty of "leaping before looking"; on the other hand "losses due to hesitation" are not especially prevalent in this industry. In any case, it is clear that much is to be gained by an appropriate *mathematical analysis* of performance and cost measures for these large systems, and that these analyses should most profitably be undertaken before major design commitments are made. This paper attempts to move in the direction of providing some tools for and insight into the design of computer networks through mathematical modeling, analysis and simulation. Frank et al.,[4] describe tools for obtaining low cost networks by choosing among topologies using computationally efficient methods from network flow theory; our approach complements theirs in that we look for closed analytic expressions where possible. Our intent is to provide understanding of the behavior and trade-offs available in some computer network situations thus creating a qualitative tool for choosing design options and not a numerical tool for choosing precise design parameters.

* This work was supported by the Advanced Research Projects Agency of the Department of Defense (DAHC15-69-C-0285).

THE ARPA EXPERIMENTAL COMPUTER NETWORK—AN EXAMPLE

The particular network which we shall use for purposes of example (and with which we are most familiar) is the Defense Department's Advanced Research Projects Agency (ARPA) experimental computer network.[2] The concepts basic to this network were clearly stated in Reference 11 by L. Roberts of the Advanced Research Projects Agency, who originally conceived this system. Reference 6, which appears in these proceedings, provides a description of the historical development as well as the structural organization and implementation of the ARPA network. We choose to review some of that description below in order to provide the reader with the motivation and understanding necessary for maintaining a certain degree of self containment in this paper.

As might be expected, the design specifications and configuration of the ARPA network have changed many times since its inception in 1967. In June, 1969, this author published a paper[8] in which a particular network configuration was described and for which certain analytical models were constructed and studied. That network consisted of nineteen nodes in the continental United States. Since then this number has changed and the identity of the nodes has changed and the topology has changed, and so on. The paper by Frank et al.,[4] published in these proceedings, describes the behavior and topological design of one of these newer versions. However, in order to be consistent with our earlier results, and since the ARPA example is intended as an illustration of an approach rather than a precise design computation, we choose to continue to study and therefore to describe the original nineteen node network in this paper.

The network provides store-and-forward communication paths between the set of nineteen computer re-

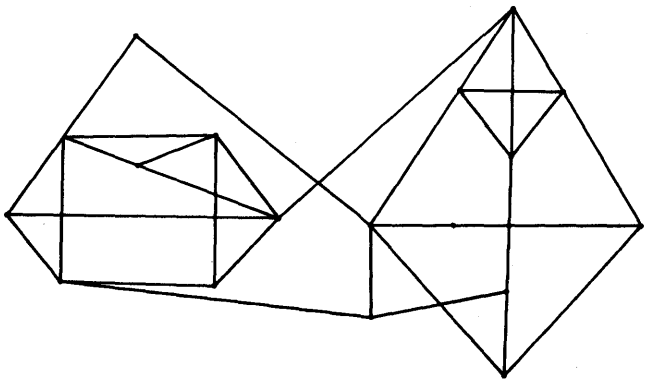


Figure 1—Configuration of the ARPA network in Spring 1969

search centers. The computers located at the various nodes are drawn from a variety of manufacturers and are highly incompatible both in hardware and software; this in fact presents the challenge of the network experiment, namely, to provide effective communication among and utilization of this collection of incompatible machines. The purpose is fundamentally for resource sharing where the resources themselves are highly specialized and take the form of unique hardware, programs, data bases, and human talent. For example, Stanford Research Institute will serve the function of network librarian as well as provide an efficient text editing system; the University of Utah provides efficient algorithms for the manipulation of figures and for picture processing; the University of Illinois will provide through its ILLIAC IV the power of its fantastic parallel processing capability; UCLA will serve as network measurement center and also provide mathematical models and simulation capability for network and time-shared system studies.

The example set of nineteen nodes is shown in Figure 1. The traffic matrix which describes the message flow required between various pairs of nodes is given in Reference 8 and will not be repeated here. An underlying constraint placed upon the construction of this network was that network operating procedures would not interfere in any significant way with the operation of the already existing facilities which were to be connected together through this network. Consequently, the message handling tasks (relay, acknowledgment, routing, buffering, etc.) are carried out in a special purpose Interface Message Processor (IMP) co-located with the principal computer (denoted HOST computer) at each of the computer research centers. The communication channels are (in most cases) 50 kilobit per second full duplex telephone lines and only the IMPs are connected to these lines through data sets. Thus the communication net consists of the lines, the IMPs and the data sets and serves as the store-and-forward system for the HOST computer network. Messages which flow between HOSTs are broken up into small entities referred to as packets (each of maximum size of approximately 1000 bits). The IMP accepts up to eight of these packets to create a maximum size message from the HOST. The packets make their way individually through the IMP network where the appropriate routing procedure directs the traffic flow. A positive acknowledgment is expected within a given time period for each inter-IMP packet transmission; the absence of an acknowledgment forces the transmitting IMP to repeat the transmission (perhaps over the same channel or some other alternate channel). An acknowledgment may not be returned for example, in the case of detected errors or for lack of buffer space in the receiving IMP. We estimate the average packet size to be 560 bits; the acknowledgment length is assumed to be 140 bits. Thus, if we assume that each packet transmitted over a channel causes the generation of a positive acknowledgment packet (the usual case, hopefully), then the average packet transmission over a line is of size 350 bits. Much of the short interactive traffic is of this nature. We also anticipate message traffic of much longer duration and we refer to this as multi-packet traffic. The average input data rate to the entire net is assumed to be 225 kilobits per second and again the reader is referred to Reference 8 for further details of this traffic distribution.

So much for the description of the ARPA network. Protocol and operating procedures for the ARPA computer network are described in References 1 and 6 in these proceedings in much greater detail. The history, development, motivation and cost of this network is described by its originator in Reference 12. Let us now proceed to the mathematical modeling, analysis and simulation of such networks.

ANALYTIC AND SIMULATION METHODS

The mathematical tools for computer network design are currently in the early stages of development. In many ways we are still at the stage of attempting to create computer network models which contain enough salient features of the network so that behavior of such networks may be predicted from the model behavior.

In this section we begin with the problem of *analysis* for a given network structure. First we review the author's earlier analytic model of communication networks and then proceed to identify those features which distinguish computer networks from strict communica-

tion networks. Some previously published results on computer networks are reviewed and then new improvements on these results are presented.

We then consider the *synthesis* and *optimization* question for networks. We proceed by first discussing the nature of the channel cost function as available under present tariff and charging structures. We consider a number of different cost functions which attempt to approximate the true data and derive relationships for optimizing the selection of channel capacities under these various cost functions. Comparisons among the optimal solutions are then made for the ARPA network.

Finally in this section we consider the *operating rules* for computer networks. We present the results of simulation for the ARPA network regarding certain aspects of the routing procedure which provide improvements in performance.

A model from queueing theory—Analysis

In a recent work[8] this author presented some computer network models which were derived from his earlier research on communication networks.[7] An attempt was made at that time to incorporate many of the salient features of the ARPA network described above into this computer network model. It was pointed out that computer networks differ from communication networks as studied in Reference 7 in at least the following features: (a) nodal storage capacity is finite and may be expected to fill occasionally; (b) channel and modem errors occur and cause retransmission; (c) acknowledgment messages increase the message traffic rates; (d) messages from HOST A to HOST B typically create return traffic (after some delay) from B to A; (e) nodal delays become important and comparable to channel transmission delays; (f) channel cost functions are more complex. We intend to include some of these features in our model below.

The model proposed for computer networks is drawn from our communication network experience and includes the following assumptions. We assume that the message arrivals form a Poisson process with average rates taken from a given traffic matrix (such as in Reference 8), where the message lengths are exponentially distributed with a mean $1/\mu$ of 350 bits (note that we are only accounting for short messages and neglecting the multi-packet traffic in this model). As discussed at length in Reference 7, we also make the independence assumption which allows a very simple node by node analysis. We further assume that a fixed routing procedure exists (that is, a unique allowable path exists from origin to destination for each origin–destination pair).

From the above assumptions one may calculate the average delay T_i due to waiting for and transmitting over the ith channel from Equation (1),

$$T_i = \frac{1}{\mu C_i - \lambda_i} \tag{1}$$

where λ_i is the average number of messages per second flowing over channel i (whose capacity is C_i bits per second). This was the appropriate expression for the average channel delay in the study of communication nets[7] and in that study we chose as our major performance measure the message delay T averaged over the entire network as calculated from

$$T = \sum_i \frac{\lambda_i}{\gamma} T_i \tag{2}$$

where γ equals the total input data rate. Note that the average on T_i is formed by weighting the delay on channel C_i with the traffic, λ_i, carried on that channel. In the study of communication nets[7] this last equation provided an excellent means for calculating the average message delay. That study went on to optimize the selection of channel capacity throughout the network under the constraint of a fixed cost which was assumed to be linear with capacity; we elaborate upon this cost function later in this section.

The computer network models studied in Reference 8 also made use of Equation (1) for the calculation of the channel delays (including queueing) where parameter choices were $1/\mu = 350$ bits, $C_i = 50$ kilobits and λ_i = average message rate on channel i (as determined from the traffic matrix, the routing procedure, and accounting for the effect of acknowledgment traffic as mentioned in feature (c) above). In order to account for feature (e) above, the performance measure (taken as the average message delay T) was calculated from

$$T = \sum_i \frac{\lambda_i}{\gamma} (T_i + 10^{-3}) \tag{3}$$

where again γ = total input data rate and the term 10^{-3} = 1 millisecond (nominal) is included to account for the assumed (fixed) nodal processing time. The result of this calculation for the ARPA network shown in Figure 1 may be found in Reference 8.

The computer network model described above is essentially the one used for calculating delays in the topological studies reported upon by Frank, et al., in these proceedings.[4]

A number of simulation experiments have been carried out using a rather detailed description of the ARPA network and its operating procedure. Some of

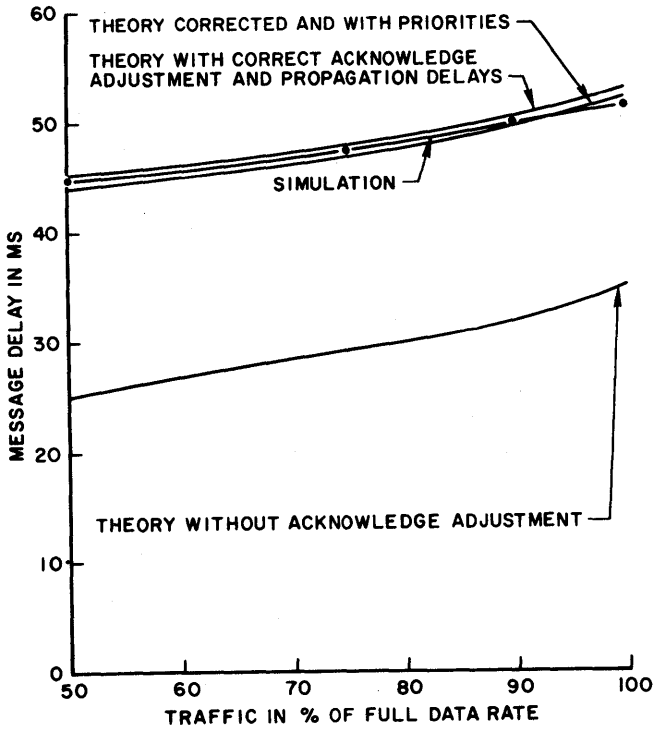


Figure 2—Comparison between theory and simulation for the ARPA network

these results were reported upon in Reference 8 and a comparison was made there between the theoretical results obtained from Equation (3) and the simulation results. This comparison is reproduced in Figure 2 where the lowest curve corresponds to the results of Equation (3). Clearly the comparison between simulation and theory is only mildly satisfactory. As pointed out in Reference 8, the discrepancy is due to the fact that the acknowledgment traffic has been improperly included in Equation (3). An attempt was made in Reference 8 to properly account for the acknowledgment traffic; however, this adjustment was unsatisfactory. The problem is that the average message length has been taken to be 350 bits and this length has averaged the traffic due to acknowledgment messages along with traffic due to real messages. These acknowledgments should not be included among those messages whose average system delay is being calculated and yet acknowledgment traffic must be included to properly account for the true loading effect in the network. In fact, the appropriate way to include this effect is to recognize that the time spent waiting for a channel is dependent upon the total traffic (including acknowledgments) whereas the time spent in transmission over a channel should be proportional to the message length of the real message traffic. Moreover, our theoretical equations have accounted only for transmission delays which come about due to the finite rate at which bits may be fed into the channel (i.e., 50 kilobits per second); we are required however to include also the propagation time for a bit to travel down the length of the channel. Lastly, an additional one millisecond delay is included in the final destination node in order to deliver the message to the destination HOST. These additional effects give rise to the following expression for the average message delay T.

$$T = \sum_i \frac{\lambda_i}{\gamma}\left(\frac{1}{\mu' C_i} + \frac{\lambda_i/\mu C_i}{\mu C_i - \lambda_i} + PL_i + 10^{-3}\right) + 10^{-3} \tag{4}$$

where $1/\mu' = 560$ bits (a real message's average length) and PL_i is the propagation delay (dependent on the channel length, L_i) for the ith channel. The first term in parentheses is the average transmission time and the second term is the average waiting time. The result of this calculation for the ARPA network gives us the curve in Figure 2 labeled "theory with correct acknowledge adjustment and propagation delays." The correspondence now between simulation and theory is unbelievably good and we are encouraged that this approach appears to be a suitable one for the prediction of computer network performance for the assumptions made here. In fact, one can go further and include the effect on message delay of the priority given to acknowledgment traffic in the ARPA network; if one includes this effect, one obtains another excellent fit to the simulation data labeled in Figure 2 as "theory corrected and with priorities."

As discussed in Reference 8 one may generalize the model considered herein to account for more general message length distributions by making use of the Pollaczek-Khinchin formula for the delay T_i of a

TABLE 1—Publicly Available Leased Transmission Line Costs from Reference 3

Speed	*Cost/mile/month (normalized to 1000 mile distance)*
45 bps	$ .70
56 bps	.70
75 bps	.77
2400 bps	1.79
41 KB	15.00
82 KB	20.00
230 KB	28.00
1 MB	60.00
12 MB	287.50

channel with capacity C_i, where the message lengths have mean $1/\mu$ bits with variance σ^2, where λ_i is the average message traffic rate and $\rho_i = \lambda_i/\mu C_i$ which states

$$T_i = \frac{1}{\mu' C_i} + \frac{\rho_i(1 + \mu^2\sigma^2)}{2(\mu C_i - \lambda_i)} \qquad (5)$$

This expression would replace the first two terms in the parenthetical expression of Equation (4); of course by relaxing the assumption of an exponential distribution we remove the simplicity provided by the Markovian property of the traffic flow. This approach, however, should provide a better approximation to the true behavior when required.

Having briefly considered the problem of analyzing computer networks with regard to a single performance measure (average message delay), we now move on to the consideration of synthesis questions. This investigation immediately leads into optimal synthesis procedures.

Optimization for various channel cost functions—Synthesis

We are concerned here with the optimization of the channel capacity assignment under various assumptions regarding the cost of these channels. This optimization must be made under the constraint of fixed cost. Our problem statement then becomes:*

Select the $\{C_i\}$ so as to minimize T

subject to a fixed cost constraint (6)

where, for simplicity, we use the expression in Equation (2) to define T.

We are now faced with choosing an appropriate cost function for the system of channels. We assume that the total cost of the network is contained in these channel costs where we certainly permit fixed termination charges, for example, to be included. In order to get a feeling for the correct form for the cost function let us examine some available data. From Reference 3 we have available the costing data which we present in Table 1. From a schedule of costs for leased communication lines available at Telpak rates we have the data presented in Table 2.

We have plotted these functions in Figure 3. We must now attempt to find an analytic function which fits cost functions of this sort. Clearly that analytic function will depend upon the rate schedule available to the computer network designer and user. Many analytic fits to this function have been proposed and in particular in Reference 3 a fit is proposed of the form:

$$\text{Cost of line} = 0.1C_i^{0.44} \quad \$/\text{mile/month} \qquad (7)$$

Based upon rates available for private line channels, Mastromonaco[10] arrives at the following fit for line costs where he has normalized to a distance of 50 miles (rather than 1000 miles in Equation (7))

$$\text{Cost of line} = 1.08C_i^{0.316} \quad \$/\text{mile/month} \qquad (8)$$

Referring now to Figure 3 we see that the mileage

TABLE 2—Estimated Leased Transmission Line Costs Based on Telpak Rates.*

Speed	*Cost (termination + mileage) /month*	*Cost/mile/month (normalized to 1000 mile distance)*
150 bps	$ 77.50 + $ .12/mile	$ .20
2400 bps	232 + .35/mile	.58
7200 bps	810 + .35/mile	1.16
19.2 KB	850 + 2.10/mile	2.95
50 KB	850 + 4.20/mile	5.05
108 KB	2400 + 4.20/mile	6.60
230.4 KB	1300 + 21.00/mile	22.30
460.8 KB	1300 + 60.00/mile	61.30
1.344 MB	500 + 75.00/mile	80.00

*These costs are, in some cases, first estimates and are not to be considered as quoted rates.

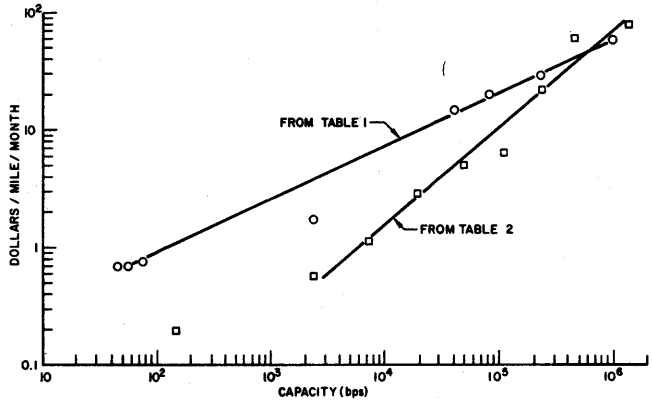


Figure 3—Scanty data on transmission line costs: $/mile/month normalized to 1000 mile distance

* The dual to this optimization problem may also be considered: "Select the $\{C_i\}$ so as to minimize cost, subject to a fixed message delay constraint." The solution to this dual problem gives the optimum C_i with the same functional dependence on λ_i as one obtains for the original optimization problem.

costs from Table 2 rise as a fractional exponent of capacity (in fact with an exponent of .815) suggesting the cost function shown in Equation (9) below

$$\text{Cost of line} = AC_i^{0.815} \quad \$/\text{mile/month} \quad (9)$$

These last three equations give the dollar cost per mile per month where the capacity C_i is given in bits per second. It is interesting to note that all three functions are of the form

$$\text{Cost of line} = AC_i^{\alpha} \quad \$/\text{mile/month} \quad (10)$$

It is clear from these simple considerations that the cost function appropriate for a particular application depends upon that application and therefore it is difficult to establish a unique cost function for all situations. Consequently, we satisfy ourselves below by considering a number of *possible* cost functions and study optimization conditions and results which follow from those cost functions. The designer may then choose from among these to match his given tariff schedule. These cost functions will form the fixed cost constraint in Equation (6). Let us now consider the collection of cost functions, and the related optimization questions.

1. *Linear cost function.* We begin with this case since the analysis already exists in the author's Reference 7, where the assumed cost constraint took the form

$$D = \sum_i d_i C_i \quad (11)$$

where D = total number of dollars available to spend on channels, d_i = the dollar cost per unit of capacity on the ith channel, and C_i once again is the capacity of the ith channel. Clearly Equation (11) is of the same form as Equation (10) with $\alpha = 1$ where we now consider the cost of all channels in the system as having a linear form. This cost function assumes that cost is strictly linear with respect to capacity; of course this same cost function allows the assumption of a constant (for example, termination charges) plus a linear cost function of capacity. This constant (termination charge) for each channel may be subtracted out of total cost, D, to create an equivalent problem of the form given in Equation (11). The constant, d_i, allows one to account for the length of the channel since d_i may clearly be proportional to the length of the channel as well as anything else regarding the particular channel involved such as, for example, the terrain over which the channel must be placed. As was done in Reference 7, one may carry out the minimization given by Equation (6) using, for example, the method of Lagrangian undetermined multipliers.[5] This procedure yields the following equation for the capacity

$$C_i = \frac{\lambda_i}{\mu} + \left(\frac{D_e}{d_i}\right) \frac{\sqrt{\lambda_i d_i}}{\sum_j \sqrt{\lambda_j d_j}} \quad (12)$$

where

$$D_e = D - \sum_i \frac{\lambda_i d_i}{\mu} > 0 \quad (13)$$

When we substitute this result back into Equation (2) we obtain that the performance measure for such a channel capacity assignment is

$$T = \frac{\bar{n}\left(\sum_i \sqrt{\lambda_i d_i/\lambda}\right)^2}{\mu D_e} \quad (14)$$

where

$$\bar{n} = \frac{\sum_i \lambda_i}{\gamma} \equiv \frac{\lambda}{\gamma} = \text{average path length} \quad (15)$$

The resulting Equation (12) is referred to as the square root channel capacity assignment; this particular assignment first provides to each channel a capacity equal to λ_i/μ which is merely the average bit rate which must pass over that channel and which it must obviously be provided if the channel is to carry such traffic. In addition, surplus capacity (due to excess dollars, D_e) is assigned to this channel in proportion to the square root of the traffic carried, hence the name. In Reference 7 the author studied in great detail the particular case for which $d_i = 1$ (the case for which all channels cost the same regardless of length) and considerable information regarding topological design and routing procedures was thereby obtained. However, in the case of the ARPA network a more reasonable choice for d_i is that it should be proportional to the length L_i of the ith channel as indicated in Equation (10) (for $\alpha = 1$) which gives the per mileage cost; thus we may take $d_i = AL_i$. This second case was considered in Reference 8 and also in Reference 9. The interpretation for these two cases regarding the desirability of concentrating traffic into a few large and short channels as well as minimizing the average length of lines traversed by a message was well discussed and will not be repeated here.

We observe in the ARPA network example since the channel capacities are fixed at 50 kilobits that there is no freedom left to optimize the choice of channel capacities; however it was shown in Reference 8 that one could take advantage of the optimization procedure in the following way: The total cost of the network

using 50 kilobit channels may be calculated. One may then optimize the network (in the sense of minimizing T) by allowing the channel capacities to vary while maintaining the cost fixed at this figure. The result of such optimization will provide a set of channel capacities which vary considerably from the fixed capacity network. It was shown in Reference 8 that one could improve the performance of the network in an efficient way by allowing that channel which required the largest capacity as a result of optimization to be increased from 50 kilobits in the fixed net to 250 kilobits. This of course increases the cost of the system. One may then provide a 250 kilobit channel for the second "most needy" channel from the optimization, increasing the cost further. One may then continue this procedure of increasing the needy channels to 250 kilobits while increasing the cost of the network and observe the way in which message delay decreases as system cost increases. It was found that natural stopping points for this procedure existed at which the cost increased rapidly without a similar sharp decrease in message delay thereby providing some handle on the cost-performance trade-off.

Since we are more interested in the difference between results obtained when one varies the cost function in more significant ways, we now study additional cost functions.

2. *Logarithmic cost functions.* The next case of interest assumes a cost function of the form

$$D = \sum_i d_i \log_e \alpha C_i \tag{16}$$

where D again is the total dollar cost provided for constructing the network, d_i is a coefficient of cost which may depend upon length of channel, α is an appropriate multiplier and C_i is the capacity of the ith channel. We consider this cost function for two reasons: first, because it has the property that the incremental cost per bit decreases as the channel size increases; and secondly, because it leads to simple theoretical results. We now solve the minimization problem expressed in Equation (6) where the fixed cost constraint is now given through Equation (16). We obtain the following equation for the capacity of the ith channel

$$C_i = \frac{\lambda_i}{\mu}\left[1 + \frac{1}{2\gamma\beta\, d_i} + \left(\frac{1}{\gamma\beta\, d_i} + \left(\frac{1}{2\gamma\beta\, d_i}\right)^2\right)^{1/2}\right] \tag{17}$$

In this solution the Lagrangian multiplier β must be adjusted so that Equation (16) is satisfied when C_i is substituted in from Equation (17). Note the unusual simplicity for the solution of C_i, namely that the channel capacity for the ith channel is *directly proportional to the traffic* carried by that channel, λ_i/μ. Contrast this result with the result in Equation (12) where we had a square root channel capacity assignment. If we now take the simple result given in Equation (17) and use it in Equation (2) to find the performance measure T we obtain

$$T = \sum_i \left\{\frac{1}{2\, d_i\beta} + \left[\frac{\gamma}{d_i\beta} + \left(\frac{1}{2\, d_i\beta}\right)^2\right]^{1/2}\right\}^{-1} \tag{18}$$

In this last result the performance measure depends upon the particular distribution of the internal traffic $\{\lambda_i/\mu\}$ through the constant β which is adjusted as described above.

3. *The power law cost function.* As we saw in Equations (7), (8), and (9) it appears that many of the existing tariffs may be approximated by a cost function of the form given in Equation (19) below.

$$D = \sum_i d_i C_i^{\alpha} \tag{19}$$

where α is some appropriate exponent of the capacity and d_i is an arbitrary multiplier which may of course depend upon the length of the channel and other pertinent channel parameters. Applying the Lagrangian again with an undetermined multiplier β we obtain as our condition for an optimal channel capacity the following non-linear equation:

$$C_i - \frac{\lambda_i}{\mu} - C_i^{(1-\alpha)/2} g_i = 0 \tag{20}$$

where

$$g_i = \left(\frac{\lambda_i}{\mu\gamma\beta\alpha\, d_i}\right)^{1/2} \tag{21}$$

Once again, β must be adjusted so as to satisfy the constraint Equation (19).

It can be shown that the left hand side of Equation (20) represents a convex function and that it has a unique solution for some positive value C_i. We assume that α is in the range

$$0 \leq \alpha \leq 1$$

as suggested from the data in Figure 3. We may also show that the location of the solution to Equation (20) is not especially sensitive to the parameter settings. Therefore, it is possible to use any efficient iterative technique for solving Equation (20) and we have found that such techniques converge quite rapidly to the optimal solution.

4. *Comparison of solutions for various cost functions.* In the last three subsections we have considered three different cost functions: the linear cost function; the logarithmic cost function; and the power law cost function. Of course we see immediately that the linear

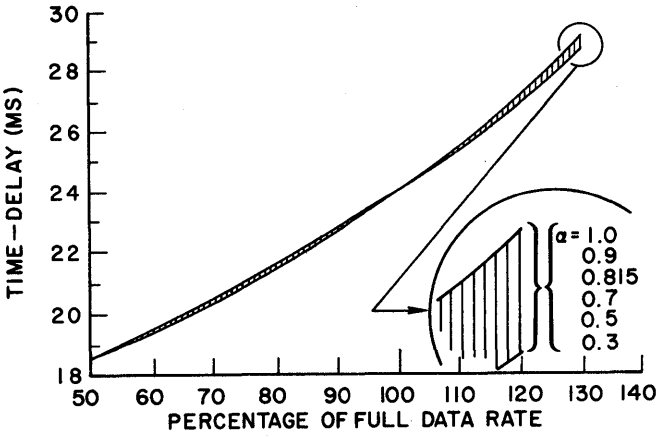


Figure 4—Average message delay at fixed cost as a function of data rate for the power law and linear cost functions

cost function is a special case $\alpha = 1$ of the power law cost function. We wish now to compare the performance and cost of computer networks under these various cost functions. We use for our example the ARPA computer network as described above.

It is not obvious how one should proceed in making this comparison. However, we adopt the following approach in an attempt to make some meaningful comparisons. We consider the ARPA network at a traffic load of 100% of the full data rate, namely 225 kilobits per second (denoted by γ_0). For the 50 kilobit net shown in Figure 1 we may calculate the line costs from Table 2 (eliminating the termination charges since we recognize this causes no essential change in our optimization procedures, as mentioned above); the resultant network cost is approximately \$579,000 per year (which we denote by D_0). Using this γ_0 and D_0 (as well as the other given input parameters) we may then carry out the optimization indicated in Equation (6) for the case of a linear cost function where $d_i = AL_i$ and A is immediately found from the mileage cost in Table 2. This calculation results in an average message delay T_0 (calculated from Equation (14)) whose value is approximately 24 milliseconds. We have now established an "operating point" for the three quantities γ_0, D_0, and T_0, whose values are 100% of full data rate, \$579,000, and 24 milliseconds, respectively.

We may now examine all of our other cost functions by forcing them to pass through this operating point. We assume $d_i = AL_i$ throughout for these calculations. Also we choose $\alpha = 1$ for the logarithmic case in Equation (16). (Note for the logarithmic and power law cases that two unknown constants, β and A, must be determined; this is now easily done if we set $T = T_0$ and $D = D_0$ for $\gamma = \gamma_0$ in each of these two cases independently.) In particular now we wish to examine the behavior of the network under these various cost functions. We do this first by fixing the cost of the network at $D = D_0$ and plotting T, the average time delay, as we vary the percentage of full data rate applied to the network; this performance is given in Figure 4 where we show the system behavior for the power law cost function and the linear cost function. The result is striking! We see that the variation in average message delay is almost insignificant as α passes through the range from 0.3 to 1.0. It appears then that the very important power law cost function may be analyzed using a linear cost function when one is interested in evaluating the average time delay at fixed cost.*

We also consider the variation of the network cost D as a function of data rate at fixed average message delay, namely $T = T_0 = 24$ milliseconds. This performance is shown in Figure 5 for all three cost functions. We note here that the linear cost function is only a fair approximation to the power law cost function over the range of α shown; the logarithmic cost function is also shown and behaves very much like the linear cost function for data rates above γ_0 but departs from that behavior for data rates below γ_0. It can be shown that the network cost, D, at fixed $T = T_0$ for the case $\alpha = 1$ (linear cost function) varies as a constant plus a linear dependence on γ. It is also of interest to cross plot the average time delay T with the network cost

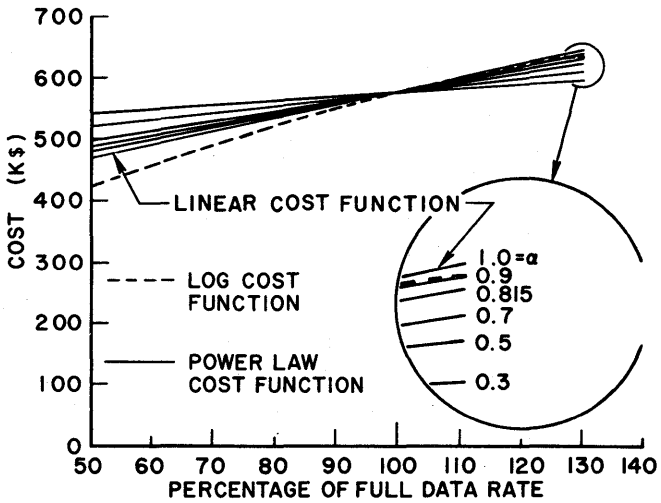


Figure 5—Network cost at fixed average message delay as a function of data rate

* The logarithm cost function is not shown in Figure 4 since the time delay is extremely sensitive to the data rate and bears little resemblance to the power law case.

D. This we do in Figure 6 for the class of power law cost functions. In Figures 6a and 6b we obtain points along the vertical and horizontal axes corresponding to fixed delay and fixed cost, respectively. These loci are obtained by varying γ and we connect the points for equal γ with straight lines as shown in the figure (however, we in no way imply that the system passes along these straight lines as both T and D are allowed to vary simultaneously). We note the increased range of D as α varies from 0.3 to 1.0, but very little change in the range of T. In Figure 6c we collect together the behavior in this plane for many values of α where the lines labeled with a particular value of α correspond to the 50% data rate case in the lower left-hand portion of the figure and to the 130% data rate case in the upper right-hand portion of the figure. From Figure 6c we clearly observe that for fixed cost the time delay range varies insignificantly as α changes (as we emphasized in discussing Figure 4). Similarly, we observe the moderate variation at fixed time delay of network cost as α ranges through its values (this we saw clearly in Figure 5).

These studies of network optimization for various cost functions need further investigation. Our aim in this section has been to exhibit some of the performance characteristics under these cost functions and to compare them in some meaningful way.

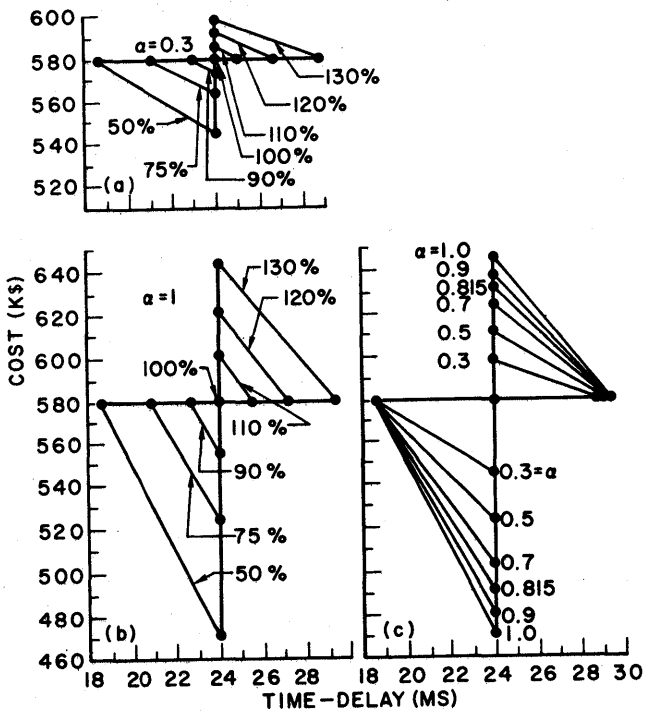


Figure 6—Locus of system performance for the power law cost function

Simulated routing in the ARPA network—Operating procedure

We have examined analysis and synthesis procedures for computer networks above. We now proceed to exhibit some properties of the network operating procedure, in particular, the message routing procedure.

The ARPA network uses a routing procedure which is local in nature as opposed to global. Some details of this procedure are available in Reference 6 in these proceedings and we wish to comment on the method used for updating the routing tables. For purposes of routing, each node maintains a list which contains for each destination an estimate of the delay a message would encounter in attempting to reach that destination node were it to be sent out over a particular channel emanating from that node; the list contains an entry for each destination and each line leaving the node in which this list is contained. Every half second (approximately) each node sends to all of its immediate neighbors a list which contains its estimate of the shortest delay time to pass to each destination; this list therefore contains a number of entries which is one less than the number of nodes in the network. Upon receiving this information from one of its neighbors, the IMP adds to this list of estimated delays a measure of the current delays in passing from itself to the neighbor from whom it is receiving this list; this then provides that IMP an estimate of the minimum delay required to reach all destinations if one traveled out over the line connected to that neighbor. The routing table for the IMP is then constructed by combining the lists of all of its neighbors into a set of columns and choosing as the output line for messages going to a particular destination that line for which the estimated delay over that line to that destination is minimum. What we have here described is essentially a periodic or *synchronous* updating method for the routing tables as currently used in the ARPA network. It has the clear advantages of providing reasonably accurate data regarding path delays as well as the important advantage of being a rather simple procedure both from an operational point of view and from an overhead point of view in terms of software costs inside the IMP program.

We suggest that a more efficient procedure in terms of routing delays is to allow *asynchronous updating*; by this we mean that routing information is passed from a node to its nearest neighbors only when significant enough changes occurr in its own routing table to warrant such an information exchange. The definition of "significant enough" must be studied carefully

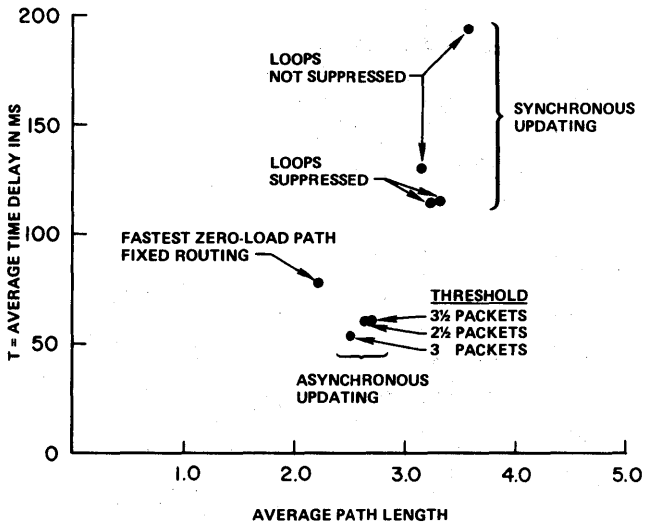


Figure 7—Comparison of synchronous and asynchronous updating for routing algorithms

but certainly implies the use of thresholds on the percentage change of estimated delays. When these thresholds are crossed in an IMP then routing information is transferred to that IMP's nearest neighbors. This asynchronous mode of updating implies a large overhead for updating and it remains to be seen whether the advantages gained through this more elaborate updating method overcome the disadvantages due to software costs and cycle-stealing costs for updating. We may observe the difference in performance between synchronous and asynchronous updating through the use of simulation as shown in Figure 7. In this figure we plot the average time delay T versus the average path length for messages under various routing disciplines. We observe immediately that the three points shown for asynchronous updating are significantly superior to those shown for synchronous updating. For a comparison we also show the result of a fixed routing algorithm which was computed by solving for the shortest delay path in an unloaded network; the asynchronous updating shows superior performance to the fixed routing procedure. Moreover, the synchronous updating shows inferior performance compared to this very simple fixed routing procedure if we take as our performance measure the average message delay.

It was observed that with synchronous updating it was possible for a message to get trapped temporarily in loops (i.e., traveling back and forth between the same pair of nodes). We suppressed this looping behavior for two synchronous updating procedures with different parameter settings and achieved significant improvement; nevertheless, this improved version remains inferior to those simulated systems with asynchronous updating. As mentioned above, asynchronous updating contains many virtues, but one must consider the overhead incurred for such a sophisticated updating procedure before it can be incorporated and expected to yield a net improvement in performance.

CONCLUSIONS

Our goal in this paper has been to demonstrate the importance of analytical and simulation techniques in evaluating computer networks in the early design stages. We have addressed ourselves to three areas of interest, namely the analysis of computer network performance using methods from queueing theory, the optimal synthesis problem for a variety of cost functions, and the choice of routing procedure for these networks. Our results show that it is possible to obtain exceptionally good results in the analysis phase when one considers the "small" packet traffic only. As yet, we have not undertaken the study of the multi-packet traffic behavior. In examining available data we found that the power law cost function appears to be the appropriate one for high-speed data lines. We obtained optimal channel capacity assignment procedures for this cost function as well as the logarithmic cost function and the linear cost function. A significant result issued from this study through the observation that the average message delay for the power law cost function could very closely be approximated by the average message delay through the system constrained by a linear cost function; this holds true in the case when the system cost is held fixed. For the fixed delay case we found that the variation of the system cost under a power law constraint could be represented by the cost variation for a linear cost constraint only to a limited extent.

In conjunction with pure analytical results it is extremely useful to take advantage of system simulation. This is the approach we described in studying the effect of routing procedures and comparing methods for updating these procedures. We indicated that asynchronous updating was clearly superior to synchronous updating except in the case where the overhead for asynchronous updating might be severe.

The results referred to above serve to describe the behavior of computer network systems and are useful in the early stages of system design. If one is desirous of obtaining numerical tools for choosing the precise design parameters of a system, then it is necessary to go to much more elaborate analytic models or else to resort to efficient search procedures (such as that

described in Reference 4) in order to locate optimal designs.

ACKNOWLEDGMENTS

The author is pleased to acknowledge Gary L. Fultz for his assistance in simulation studies as well as his contributions to loop suppression in the routing procedures; acknowledgment is also due to Ken Chen for his assistance in the numerical solution for the performance under different cost function constraints.

REFERENCES

1 S CARR S CROCKER V CERF
Host to host communication protocol in the ARPA network
These proceedings

2 P A DICKSON
ARPA network will represent integration on a large scale
Electronics pp 131–134 September 30 1968

3 R G GOULD
Comments on generalized cost expressions for private-line communications channels
IEEE Transactions on Communication Technology V Com-13 No 3 pp 374–377 September 1965
also
R P ECKHERT P M KELLY
A program for the development of a computer resource sharing network
Internal Report for Kelly Scientific Corp Washington D C February 1969

4 H FRANK I T FRISCH W CHOU
Topological considerations in the design of the ARPA computer network
These proceedings

5 F B HILDEBRAND
Methods of applied mathematics
Prentice-Hall Inc Englewood Cliffs N J 1958

6 F E HEART R E KAHN S M ORNSTEIN W R CROWTHER D C WALDEN
The interface message processor for the ARPA network
These proceedings

7 L KLEINROCK
Communication nets; stochastic message flow and delay
McGraw-Hill New York 1964

8 L KLEINROCK
Models for computer networks
Proc of the International Communications Conference pp 21–9 to 21–16 University of Colorado Boulder June 1969

9 L KLEINROCK
Comparison of solutions methods for computer network models
Proc of the Computers and Communications Conference Rome New York September 30–October 2 1969

10 F R MASTROMONACO
Optimum speed of service in the design of customer data communications systems
Proc of the ACM Symposium on the Optimization of Data Communications Systems pp 127–151 Pine Mountain Georgia October 13–16 1969

11 L G ROBERTS
Multiple computer networks and intercomputer communications
ACM Symposium on Operating Systems Principles Gatlinburg Tennessee October 1967

12 L G ROBERTS B D WESSLER
Computer network developments to achieve resource sharing
These proceedings

Topological considerations in the design of the ARPA computer network*

by H. FRANK, I. T. FRISCH, and W. CHOU

Network Analysis Corporation
Glen Cove, New York

INTRODUCTION

The ARPA Network will provide store-and-forward communication paths between a set of computer centers distributed across the continental United States. The message handling tasks at each node in the network are performed by a special purpose Interface Message Processor (IMP) located at each computer center. The centers will be interconnected through the IMPs by fully duplex telephone lines, of typically 50 kilobit/sec capacity.

When a message is ready for transmission, it will be broken up into a set of packets, each with appropriate header information. Each packet will independently make its way through the network to its destination. When a packet is transmitted between any pair of nodes, the transmitting IMP must receive a positive acknowledgement from the receiving IMP within a given interval of time. If this acknowledgement is not received, the packet will be retransmitted, either over the same or a different channel depending on the network routing doctrine being employed.

One of the design goals of the system is to achieve a response time of less than 0.2 seconds for short messages. A measure of the efficiency with which this criterion is met is the cost per bit of information transmitted through the network when the total network traffic is at the level which yields 0.2 second average time delay. The goal of the network design is to achieve the required response time with the least possible cost per bit. The final network design is subject to a number of additional constraints. It must be reliable, it must have reasonably flexible capacity in order to accommodate variations in traffic flow without significant degradation in performance, and it must be neatly expandable so that additional nodes and links can be added at later dates. The sequence and allowable variations with which the nodes are added to the network must also be taken into account. At any stage in the evolution of the network, there must be at least one communication path between any pair of nodes that have already been activated. In order to achieve a reasonable level of reliability, the network must be designed so that at least two nodes and/or links must fail before the network becomes disconnected.

To plan the orderly growth of the network, it is necessary to predict the behavior of proposed network designs. To do this, traffic flows must be projected and network routing procedures specified. The time delay analysis problem has been studied by Kleinrock[1,2] who considered several mathematical models of the ARPA Network. Kleinrock's comparison of his analysis with computer simulations indicates that network behavior can be qualitatively predicted with reasonable confidence. However, additional study in this area is needed before all the significant parameters which describe the system can be incorporated into the model. For the present, it appears that a combination of analysis and simulation can best be applied to determine a specific network's behavior.

Even if a proposed network can be accurately analyzed, the most economical networks which satisfy all of the constraints are not easily found. This is because of the enormous number of combinations of links that can be used to connect a relatively small number of nodes. It is not possible to examine even a small fraction of the possible network topologies that might lead to economical designs. In fact, the direct enumeration of all such configurations for a twenty node network is beyond the capabilities of the most powerful present day computer.

* This work was supported by the Advanced Research Projects Agency of the Department of Defense (Contract No. DAHC15-70-C-0120).

TOPOLOGICAL OPTIMIZATION

As part of NAC's study of computer network design, a computer program was developed to find low cost topologies which satisfy the constraints on network time delay, reliability, congestion, and other performance parameters. This program is structured to allow the network designer to rapidly investigate the tradeoffs between average time delay per message, network cost, and other factors of interest.

The inputs to the program are:

1. Existing network configuration (i.e., lines and nodes already installed and ordered)
2. Estimated traffic between nodes
3. Maximum average delay desired for short messages

In addition, the user may specify to the program a maximum cost that no network design will be allowed to exceed.

The output of the program is a sequence of low cost networks. Each network is identified by the following information:

1. Network topology
2. Cost per month
3. Maximum throughput
4. Estimated average traffic
5. Message cost per megabit at maximum throughput
6. Average message delay for short messages

Each acceptable network design also conforms to the standard that at least two nodes and/or links must fail before all communication paths between any pair of nodes are disrupted.

APPROACH

The general design problem as stated above is similar to other network design problems for which computationally practical solutions have recently been obtained. These problems include the minimum cost design of survivable networks,[3] the minimum cost selection and interconnection of Telpaks in telephone networks,[4] the design of offshore natural gas pipeline networks,[5] and the classical Traveling Salesman problem.[6] These problems have long resisted exact solution; however, recent work on approximate methods has been extremely successful and has led to efficient methods of finding low cost solutions in practical computation times.

The design philosophy

By a "feasible" solution, we mean one which satisfies all of the network constraints. By an "optimal" network, we mean the feasible network with the least possible cost. Our goal is to develop a method that can handle realistically large problems in a reasonable computation time and which can find feasible solutions with costs close to optimal.

The method to be used has two main parts called the *starting routine* and the *optimizing routine*. The starting routine generates a feasible solution. The optimizing routine then examines networks derived from this starting network by means of local transformations applied to the network topology. When a feasible network with lower cost is found, it is adopted as a new starting network and the process is continued. In this way, a feasible network is eventually reached whose cost cannot be reduced by applying additional local transformations of the type being considered. Such a network is called a *locally optimum* network.

Once a locally optimum network is found, the entire procedure is repeated by again using the starting routine. The starting routine may incorporate suggestions made by a human designer. For example, the present tentative configurations for the ARPA Network have been used. Alternatively, if desired, the starting routine may generate feasible networks without such advice. At the present time, our starting routine is capable of generating about 100,000 low cost networks.

By finding local optima from different starting networks, a variety of solutions can be generated. Figure 1 shows a diagrammatic representation of the process.

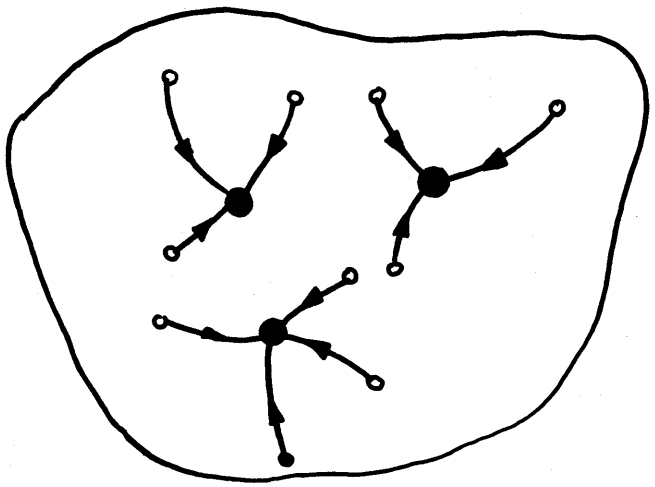


Figure 1—Diagrammatic representation of the optimization procedure

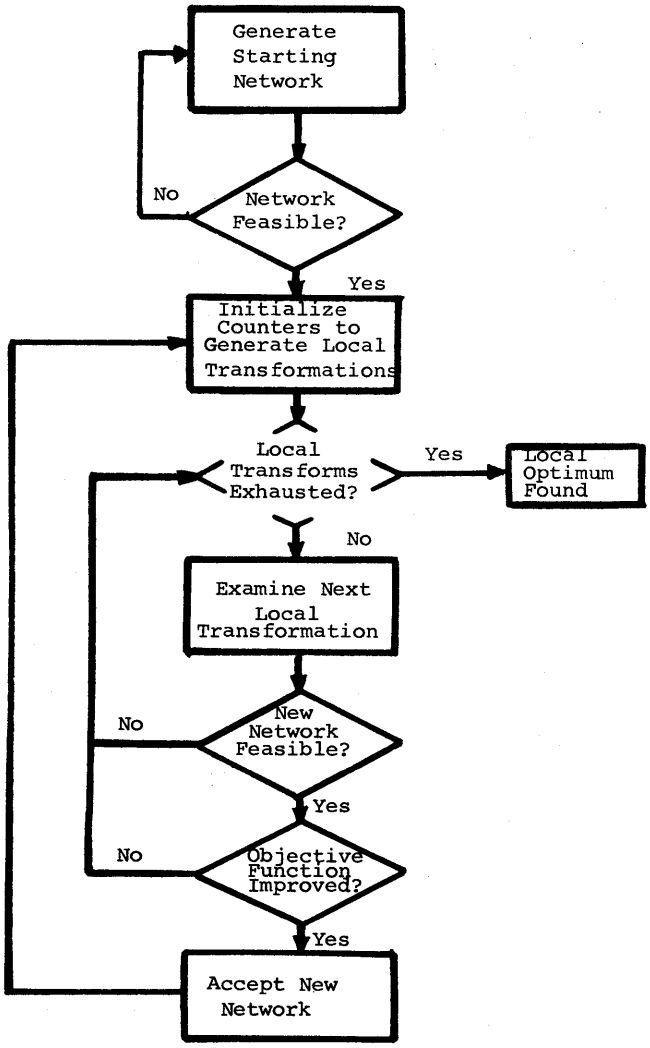


Figure 2—Block diagram of optimization procedure

The space of feasible solutions is represented by the area enclosed by the outer border of the figure; starting solutions are represented by light circles and local optima by dark circles. The practicality of the approach is based on the assumption that with a high probability some of the local optima found are close in cost to the global optimum. Naturally, this assumption is sensitive to the particular transformation used in the optimizing routine. A block diagram of the optimization procedure is shown in Figure 2.

Local transformations

A local transformation on a network is generated by identifying a set of links, removing these links, and adding a new set to the network. The method of selection of the number and location of the links to be removed and added determines the usefulness of the transformation and its applicability to the problem in hand. For example, in the problem of economically designing offshore natural gas pipeline networks, dramatic cost reductions were achieved by removing and adding one link at a time.[5] On the other hand, in a problem of the minimum cost design of survivable networks, the most useful link exchange consisted of removing and adding two links at a time.[3] In general, it is not necessary that the same number of links be added and removed during each application of the transformation.

DESIGN CONSTRAINTS

The preceding section has a given general approach for the design of low cost feasible networks. To implement this approach, a number of specific problems must be considered. These include:

1. The distribution of network traffic.
2. Network Route Selection.
3. Link capacity assignment.
4. Node and Link Time Delays.

Distribution of traffic

At the present time, it is difficult to estimate the precise magnitude and distribution of the Host-to-Host traffic. However, one design goal is that the amount of flow that can be transmitted between nodes should not significantly vary with the locations of sender and receiver. Hence, two users several thousand miles apart should receive the same service as two users several hundred miles apart. A reasonable requirement is therefore that the network be designed so that it can accommodate equal traffic between all pairs of nodes. However, it is known that certain nodes have larger traffic requirements to and from the University of Illinois' Illiac IV than to other nodes. Consequently, information of this type is incorporated into the model.

The magnitude of the network traffic is treated as variable. A "base" traffic requirement of $500 \cdot n$ bits per second (n is a positive real number) between all nodes is assumed. An additional $500 \cdot n$ bits per second is then added to and from the University of Illinois (node No. 9) and nodes 4, 5, 12, 18, 19, and 20. The base traffic is used to determine the flows in each link and the link capacities as discussed in the following sections. n is then increased until the average time delay exceeds .2 seconds. The average number of bits per second per

node at average delay equal .2 seconds is taken as a measure of performance and the corresponding cost per bit is taken as a measure of efficiency of the network.

Route selection

In order to avoid the prohibitively long computation times required to analyze dynamic routing strategies, a fixed routing procedure is used. This procedure is similar to the one which will be used in the operating network but it has the advantage that it can be readily incorporated into analysis procedures which do not depend on simulation.

The routing procedure is determined by the assumption that for each message a *path which contains the fewest number of intermediate* nodes from origin to destination is most desirable.* Given a proposed network topology and traffic matrix, routes are determined as follows: For each i ($i = 1, 2, \cdots, N = 20$):

1. With node i as an initial node, use a labelling procedure[7] to generate all paths containing the *fewest* number of intermediate nodes, to all nodes which have non-zero traffic from node i. Such paths are called *feasible paths.*
2. If node i has non-zero traffic to node j ($j = 1, 2, \cdots, N, j \neq i$) and the feasible paths from i to j contain more than seven nodes, the topology is considered infeasible.
3. Nodes are grouped as follows:
   (a) All nodes connected to node i.
   (b) All nodes connected to node i by a feasible path with one intermediate node.
   (c) All nodes connected to node i by a feasible path with two intermediate nodes.
   (d) — — — — — — — — — —
   (e) — — — — — — — — — —
   (f) All nodes connected to node i by a feasible path with five intermediate nodes.

Traffic is first routed from node i to any node j which is directly connected to i over link (i, j). Consequently, after this stage, some flows have been assigned to the network. Each node in group (b) is then considered. For any node j in this group, all feasible paths from i to j are examined, and the maximum flow thus far assigned in any link in each such path is found. All paths with the smallest maximum flow are then considered. The path whose total length is minimum is then selected and all traffic originating at i and destined for j is routed over this path.* All nodes in group (b) are treated in this matter. The same procedure is then applied to all nodes in group (c), (d), (e) and (f) in that order.

Capacity assignment

Link capacities could be assigned prior to routing. Then after route selection, if the flow in any link exceeds its assigned capacity, the network would be considered infeasible. On the other hand, link capacities may be assigned *after* all traffic is routed; we adopt this approach. The capacity of each link is chosen to be the least expensive option available from AT&T which satisfies the flow requirement. The line options which are presently being considered are: 50,000 bits/sec (bps), 108,000 bps, 230,400 bps, and 460,000 bps. Monthly link costs are the sum of a fixed terminal charge and a linear cost per mile. Thus, to satisfy a requirement of 85,000 bps, depending on the length of the link it is sometimes cheaper to use two 50,000 bps parallel links and sometimes cheaper to use a single 108,000 bps link.

The following line options and costs have been investigated:

Type	*Speed*	*Cost Per Month*
Full Group (303 data set)	50 KB	\$850 + \$4.20/mile
Full Group (304 data set)**	108 KB	\$2400 + \$4.20/mile
Telpak C	230.4 KB	\$1300 + \$21.00/mile
Telpak D	460 KB	\$1300 + \$60.00/mile

Link and node delays

Response time T is defined as the average time a message takes to make its way through the network from its origin to its destination. Short messages are considered to correspond to a single packet which may be as long as 1008 bits or as short as few bits, plus the header. If T_i is the mean delay time for a packet passing through the ith link, then

$$T = r^{-1} \sum_{i=1}^{M} y_i T_i,$$

* A node $j \neq s, t$ is called an *intermediate node* with respect to a message with origin s and destination t if the path from s to t over which the message is transmitted contains node j.

*It is also possible to divide the traffic from i to j and send it over more than one feasible path, but for uniform traffic this is not an important factor.

**Not a standard AT&T offering.

where r is the total IMP-to-IMP traffic rate, y_i is the average traffic rate in the ith link, and M is the total number of links. T_i can be approximated with the Pollaczak-Khinchin formula as:

$$T_i = \frac{1}{\mu C_i}\left[1 + \frac{y_i(1 + a^2)}{2(\mu C_i - y_i)}\right]$$

where $1/\mu$ is the average packet length (in bits), C_i is the capacity of the ith link (in bits/second), a is the coefficient of variance for the packet length.

These parameters are evaluated as follows:

1. r is the sum of all elements in the traffic matrix after each element has been adjusted to include headers, parity check and requests for next message (RFNM).
2. y_i is determined by the routing strategy.
3. In calculating $1/\mu$, we consider three kinds of packets: (a) packets generated by short messages and all other packets (except RFNM's) with length less than 1008 bits; (b) full length packets of 1008 bits belonging to long messages; (c) RFNM's.

It is assumed that the packets of part (a) are uniformly distributed with mean length equal to 560 bits. The packet length for part (b) is a constant equal to 1008 bits. The average packet length is then calculated by first estimating the average number of packets with 1008 bits. It is assumed that each long message consists of an average of 4 packets. In many of our computations, we assume that 80% of the messages are short. The number of RFNM packets can then be estimated. Finally, since the average length of each type of packet is known and the number of each type of packet has been estimated, the average packet length can be estimated.

4. y_i is adjusted to include the increased traffic due to acknowledgments. C_i is then selected as already described.
5. The larger the value of a, the larger the delay time. For the exponential distribution $a = 1$; for a constant, $a = 0$; and for many distributions $0 < a < 1$. Since it is reasonable to assume that the packet length distribution being considered is very close to the combination of a uniform distribution and a constant, the value of a should be less than one. To avoid underestimating T, a is set equal to one in all calculations.

The above analysis is based on the assumption that the number of available buffers is unlimited. When the traffic is low, this assumption is very accurate. For high traffic, adjustments to account for the limitation of buffer space are necessary.

There are two roles for buffers in an IMP; one for reassembling messages destined for that IMP's Host and the other for store-and-forward traffic. At the present time, about one-half of the IMP's core is used for the operating program. The remainder contains about 84 buffers each of which can store a single packet. Up to $\frac{2}{3}$ of the buffers may be used for reassembly. Buffers not used for reassembly are available for store-and-forward traffic. When no buffer is available for reassembly, any arriving packet which requires reassembly but does not belong to any message in the process of reassembly will be discarded and no acknowledgment returned to the transmitting IMP. This packet must then be retransmitted, and the effective traffic in the link is therefore increased. In addition, each time a packet is retransmitted, its delay time is not only increased by the extra waiting and transmitting time, but also by the 100 ms time-out period. To account for these factors, an upper bound on the probability that no buffer is available is calculated for each IMP. The traffic between IMPs is then increased and extra delay time for the retransmitted packets is calculated. The increase in delay time is then averaged over all the packets.

When no buffer is available for store-and-forward traffic, all incoming links become inactive. Effectively, the average usable capacities of these links is lower than their actual capacities. The probability that no buffer is available for store-and-forward traffic is set equal to the average of an upper bound and a lower bound; the upper bound is calculated by assuming that the ratio of flow to capacity of each link into the IMP is equal to the maximum ratio for all links at that node while the lower bound is found by assuming that the ratio of flow to capacity for each link is equal to the minimum such ratio. Link capacities are then reduced to include this effect and the response time is then recalculated. An example of the effect of the above assumptions is shown in Figure 4. Figure 4 relates average time delay and throughput per node for the network shown in Figure 3. Two curves are shown. One is obtained by assuming that there are an infinite number of buffers at each node. The second curve is obtained by using the actual buffer limitations of the ARPA network.

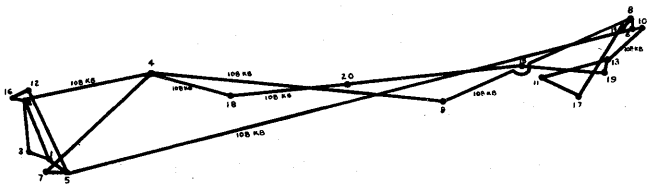


Figure 3

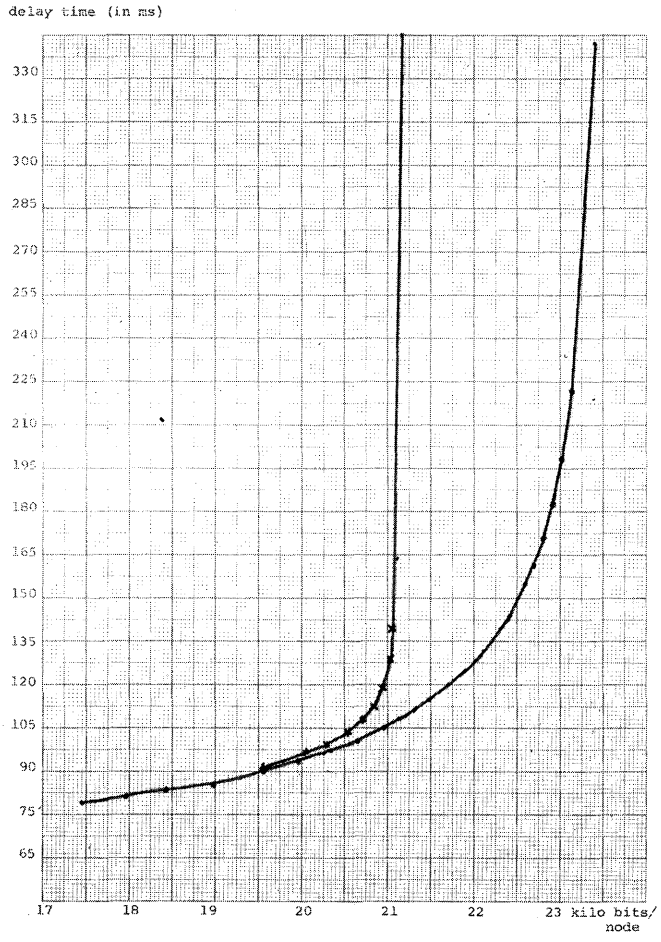


Figure 4

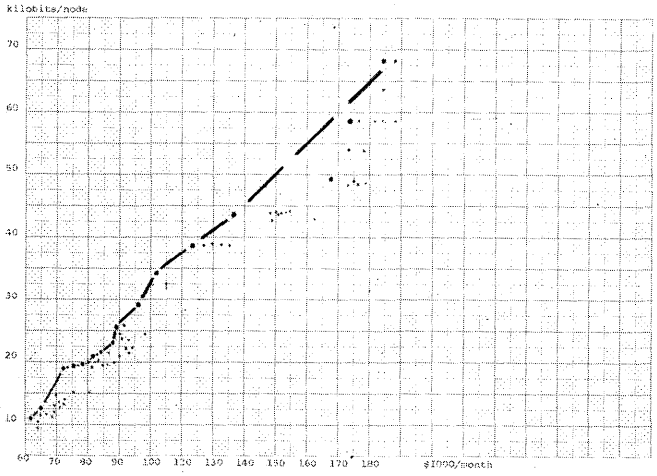


Figure 5

PRELIMINARY COMPUTATIONAL RESULTS

The optimization procedures were employed to design many thousand twenty node networks. The parameters of the best of these networks were then plotted as scatter diagrams as indicated in Figure 5. The coordinate of the horizontal axis on the graph is cost in dollars. The coordinate of the vertical axis is the average throughput per node* in bits per second for a specified distribution of traffic. The graph shown is for an average message delay of .2 seconds for short messages. Each point in the graph corresponds to a network generated, evaluated, and optimized by the computer.

Interpretation of results

Consider any point P_1 corresponding to a network N_1. Draw a horizontal line starting at P_1 to the right of P_1 and a vertical line down from P_1. Any point say P_2 which falls within the quadrant defined by the two lines is said to be *dominated by* P_1, since in a sense, network N_1 is "better than" network N_2. Similarly N_1 is said to be a dominant network. That is, for the same delay N_1 provides at least as much throughput as N_2 at no higher cost. Horizontal and vertical lines can be drawn through certain points $P_1, \cdots, P_n$ so that all other points are dominated by at least one of these. $P_1, \cdots, P_n$ thus represent, in one sense, the best networks.

One must be cautious, however, in that a network which is dominant for one time delay may not be dominant for another. Many networks with this property have been found in our studies.

Furthermore, in some cases a network may be dominated but might still be preferable to the network which dominates it because of other factors such as the order of leasing lines and plans for future growth. As an example, P_1 is a dominant point and yet there are many points which it dominates which are very close to it and might well be preferable.

Some other conclusions can be drawn from the graphs. Examining the set of dominant points it appears that there are significant savings due to economies of scale in the range of costs of \$64,000 to \$80,000. That is, small increases in cost yield large gains in throughput. Similar savings are observed in the \$90,000–\$100,000 cost range for average throughputs in the 30,000 bits/second range. These savings are due to the utilization of 108 kilobit lines which have the same line cost as 50 kilobit lines but a higher data set cost. This means that for a modest additional cost, the capacities of cross country lines can be more than doubled. To see

* throughput is the average number of bits/second *out* of each node.

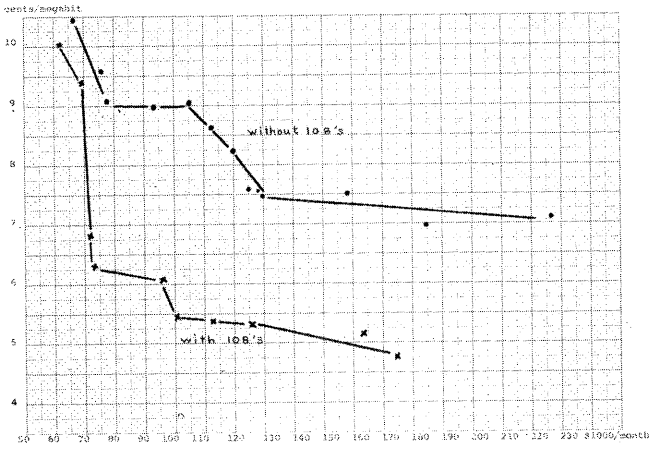


Figure 6

the effect of eliminating the 108 kilobit line option (which is not a standard AT&T offering), the cost per megabit of transmitted data is plotted against the total monthly line cost in Figure 6 for low cost networks designed with and without this option. Each point in this figure represents a feasible network. The points are connected by straight lines for visual convenience.

Additional investigations are presently under way to better understand the relationship between cost, delay and throughput, and the effect of the number of nodes on these parameters. Furthermore, alternative routing schemes will be considered as well as the cost-throughput tradeoffs that can be obtained by increasing the number of buffers at appropriate nodes.

REFERENCES

1 L KLEINROCK
Models for computer networks
Proceedings of the International Conference on Communications pp 21.9–21.16 June 1969

2 L KLEINROCK
Analytic and simulation methods in computer network design.
See paper this conference

3 K STEIGLITZ P WEINER D KLEITMAN
Design of minimum cost survivable networks
IEEE Transactions on Circuit Theory 1970

4 B ROTHFARB M GOLDSTEIN
Unpublished work

5 H FRANK B ROTHFARB D KLEITMAN K STEIGLITZ
Design of economical offshore natural gas pipeline networks
Office of Emergency Preparedness Report No R-1 Washington D C January 1969

6 S LIN
Computer solutions of the traveling salesman problem
Bell System Tech Journal Vol 44 No 10 pp 2245–2269 December 1965

7 H FRANK I T FRISCH
Communication, transmission, and transportation networks
Addison–Wesley 1971

HOST-HOST communication protocol in the ARPA network*

by C. STEPHEN CARR

University of Utah
Salt Lake City, Utah

and

STEPHEN D. CROCKER and VINTON G. CERF

University of California
Los Angeles, California

INTRODUCTION

The Advanced Research Projects Agency (ARPA) Computer Netwok (hereafter referred to as the "ARPA network") is one of the most ambitious computer networks attempted to date.[1] The types of machines and operating systems involved in the network vary widely. For example, the computers at the first four sites are an XDS 940 (Stanford Research Institute), an IBM 360/75 (University of California, Santa Barbara), an XDS SIGMA-7 (University of California, Los Angeles), and a DEC PDP-10 (University of Utah). The only commonality among the network membership is the use of highly interactive time-sharing systems; but, of course, these are all different in external appearance and implementation. Furthermore, no one node is in control of the network. This has insured generality and reliability but complicates the software.

Of the networks which have reached the operational phase and been reported in the literature, none have involved the variety of computers and operating systems found in the ARPA network. For example, the Carnegie-Mellon, Princeton, IBM network consists of 360/67's with identical software.[2] Load sharing among identical batch machines was commonplace at North American Rockwell Corporation in the early 1960's. Therefore, the implementers of the present network have been only slightly influenced by earlier network attempts.

However, early time-sharing studies at the University of California at Berkeley, MIT, Lincoln Laboratory, and System Development Corporation (all ARPA sponsored) have had considerable influence on the design of the network. In some sense, the ARPA network of time-shared computers is a natural extension of earlier time-sharing concepts.

The network is seen as a set of data entry and exit points into which individual computers insert messages destined for another (or the same) computer, and from which such messages emerge. The format of such messages and the operation of the network was specified by the network contractor (BB&N) and it became the responsibility of representatives of the various computer sites to impose such additional constraints and provide such protocol as necessary for users at one site to use resources at foreign sites. This paper details the decisions that have been made and the considerations behind these decisions.

Several people deserve acknowledgment in this effort. J. Rulifson and W. Duvall of SRI participated in the early design effort of the protocol and in the discussions of NIL. G. Deloche of Thomson-CSF participated in the design effort while he was at UCLA and provided considerable documentation. J. Curry of Utah and P. Rovner of Lincoln Laboratory reviewed the early design and NIL. W. Crowther of Bolt, Beranek and Newman contributed the idea of a virtual net. The BB&N staff provided substantial assistance and guidance while delivering the network.

We have found that, in the process of connecting machines and operating systems together, a great deal of rapport has been established between personnel at

*This research was sponsored by the Advanced Research Projects Agency, Department of Defense, under contracts AF30(602)-4277 and DAHC15-69-C-0285.

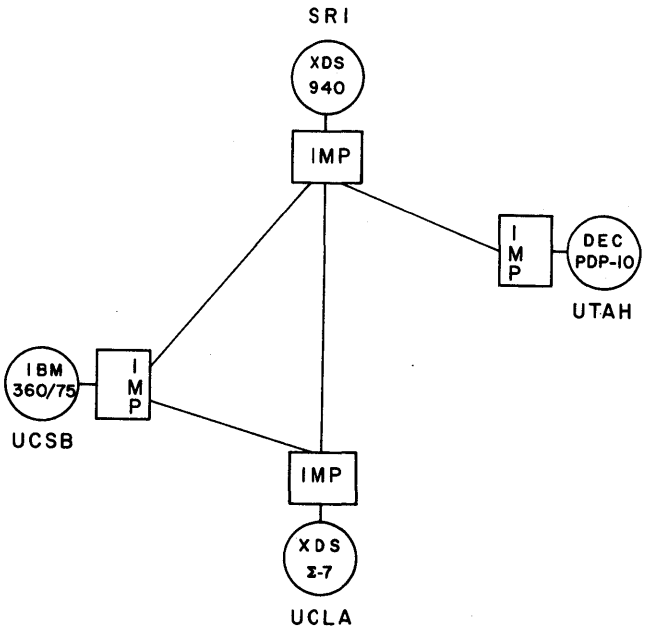


Figure 1—Initial network configuration

the various network node sites. The resulting mixture of ideas, discussions, disagreements, and resolutions has been highly refreshing and beneficial to all involved, and we regard the human interaction as a valuable by-product of the main effort.

THE NETWORK AS SEEN BY THE HOSTS

Before going on to discuss operating system communication protocol, some definitions are needed.

A *HOST* is a computer system which is part of the network.

An *IMP* (Interface Message Processor) is a Honeywell DDP–516 computer which interfaces with up to four HOSTs at a particular site, and allows HOSTs access into the network. The configuration of the initial four-HOST network is given in Figure 1. The IMPs form a store-and-forward communications network. A companion paper in these proceedings covers the IMPs in some detail.[3]

A *message* is a bit stream less than 8096 bits long which is given to an IMP by a HOST for transmission to another HOST. The first 32 bits of the message are the *leader*. The leader contains the following information:

(a) HOST
(b) Message type
(c) Flags
(d) Link number

When a message is transmitted from a HOST to its IMP, the HOST field of the leader names the receiving HOST. When the message arrives at the receiving HOST, the HOST field names the sending HOST.

Only two message types are of concern in this paper. Regular messages are generated by a HOST and sent to its IMP for transmission to a foreign HOST. The other message type of interest is a RFNM (Request-for-Next-Message). RFNMs are explained in conjunction with links.

The flag field of the leader controls special cases not of concern here.

The link number identifies over which of 256 logical paths (links) between the sending HOST and the receiving HOST the message will be sent. Each link is unidirectional and is controlled by the network so that no more than one message at a time may be sent over it. This control is implemented using RFNM messages. After a sending HOST has sent a message to a receiving HOST over a particular link, the sending HOST is prohibited from sending another message over that same link until the sending HOST receives a RFNM. The RFNM is generated by the IMP connected to the receiving HOST, and the RFNM is sent back to the sending HOST after the message has entered the receiving HOST. It is important to remember that there are 256 links in each direction and that no relationship among these is imposed by the network.

The purpose of the link and RFNM mechanism is to prohibit individual users from overloading an IMP or a HOST. Implicit in this purpose is the assumption that a user does not use multiple links to achieve a wide band, and to a large extent the HOST–HOST protocol cooperates with this assumption. An even more basic assumption, of course, is that the network's load comes from some users transmitting sequences of messages rather than many users transmitting single messages coincidently.

In order to delimit the length of the message, and to make it easier for HOSTs of differing word lengths to communicate, the following formatting procedure is used. When a HOST prepares a message for output, it creates a 32-bit leader. Following the leader is a binary string, called *marking*, consisting of an arbitrary number of zeroes, followed by a one. Marking makes it possible for the sending HOST to synchronize the beginning of the text of a message with its word boundaries. When the last bit of a message has entered an IMP, the hardware interface between the IMP and HOST appends a one followed by enough zeroes to make the message length a multiple of 16 bits. These appended bits are called *padding*. Except for the marking and padding, no limitations are placed on the text of a message. Figure 2 shows a typical message sent by a 24-bit machine.

DESIGN CONCEPTS

The computers participating in the network are alike in two important respects: each supports research inde-

pendent of the network, and each is under the discipline of a time-sharing system. These facts contributed to the following design philosophy.

First, because the computers in the network have independent purposes, it is necessary to preserve decentralized administrative control of the various computers. Since all of the time-sharing supervisors possess elaborate and definite accounting and resource allocation mechanisms, we arranged matters so that these mechanisms would control the load due to the network in the same way they control locally generated load.

Second, because the computers are all operated under time-sharing disciplines, it seemed desirable to facilitate basic interactive mechanisms.

Third, because this network is used by experienced programmers it was imperative to provide the widest latitude in using the network. Restrictions concerning character sets, programming languages, etc., would not be tolerated and we avoided such restrictions.

Fourth, again because the network is used by experienced programmers, it was felt necessary to leave the design open-ended. We expect that conventions will arise from time to time as experience is gained, but we felt constrained not to impose them arbitrarily.

Fifth, in order to make network participation comfortable, or in some cases, feasible, the software interface to the network should require minimal surgery on the HOST operating system.

Finally, we accepted the assumption stated above that network use consists of prolonged conversations instead of one-shot requests.

Those considerations led to the notions of connections, a Network Control Program, a control link, control commands, sockets, and virtual nets.

A *connection* is an extension of a link. A connection connects two processes so that output from one process is input to the other. Connections are simplex, so two connections are needed if two processes are to converse in both directions.

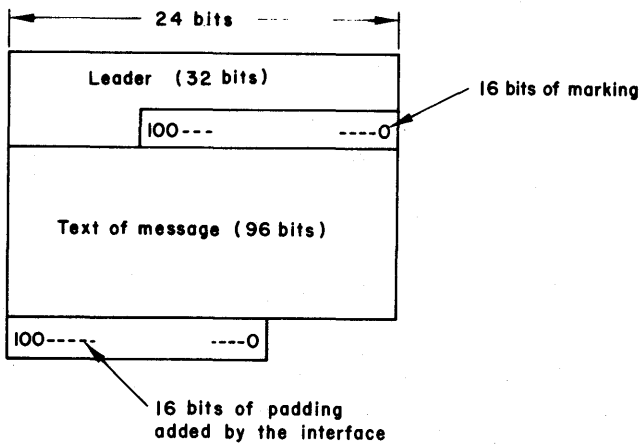


Figure 2—A typical message from a 24-bit machine

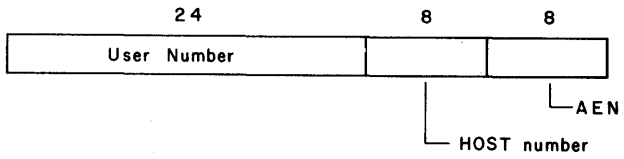


Figure 3—A typical socket

Processes within a HOST communicate with the network through a *Network Control Program* (NCP). In most HOSTs, the NCP will be part of the executive, so that processes will use system calls to communicate with it. The primary function of the NCP is to establish connections, break connections, switch connections, and control flow.

In order to accomplish its tasks, a NCP in one HOST must communicate with a NCP in another HOST. To this end, a particular link between each pair of HOSTs has been designated as the *control link*. Messages received over the control link are always interpreted by the NCP as a sequence of one or more *control commands*. As an example, one of the kinds of control commands is used to assign a link and initiate a connection, while another kind carries notification that a connection has been terminated. A partial sketch of the syntax and semantics of control commands is given in the next section.

A major issue is how to refer to processes in a foreign HOST. Each HOST has some internal naming scheme, but these various schemes often are incompatible. Since it is not practical to impose a common internal process naming scheme, an intermediate name space was created with a separate portion of the name space given to each HOST. It is left to each HOST to map internal process identifiers into its name space.

The elements of the name space are called *sockets*. A socket forms one end of a connection, and a connection is fully specified by a pair of sockets. A socket is specified by the concatenation of three numbers:

(a) a user number (24 bits)
(b) a HOST number (8 bits)
(c) AEN (8 bits)

A typical socket is illustrated in Figure 3.

Each HOST is assigned all sockets in the name space which have field (b) equal to the HOST's own identification.

A socket is either a *receive socket* or a *send socket*, and is so marked by the low-order bit of the AEN (0 = receive, 1 = send). The other seven bits of the

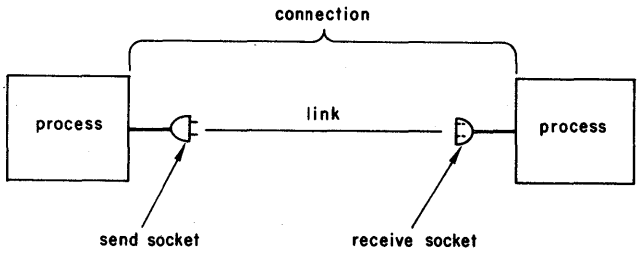


Figure 4—The relationship between sockets and processes

AEN simply provide a sizable population of sockets for each user number at each HOST. (AEN stands for "another eight-bit number".)

Each user is assigned a 24-bit user number which uniquely identifies him throughout the network. Generally this will be the 8-bit HOST number of his home HOST, followed by 16 bits which uniquely identify him at that HOST. Provision can also be made for a user to have a user number not keyed to a particular HOST, an arrangement desirable for mobile users who might have no home HOST or more than one home HOST. This 24-bit user number is then used in the following manner. When a user signs onto a HOST, his user number is looked up. Thereafter, each process the user creates is tagged with his user number. When the user signs onto a foreign HOST via the network, his same user number is used to tag processes he creates in that HOST. The foreign HOST obtains the user number either by consulting a table at login time, as the home HOST does, or by noticing the identification of the caller. The effect of propagating the user's number is that each user creates his own *virtual net* consisting of processes he has created. This virtual net may span an arbitrary number of HOSTs. It will thus be easy for a user to connect his processes in arbitrary ways, while still permitting him to connect his processes with those in other virtual nets.

The relationship between sockets and processes is now describable (see Figure 4). For each user number at each HOST, there are 128 send sockets and 128 receive sockets. A process may request from the local NCP the use of any one of the sockets with the same user number; the request is granted if the socket is not otherwise in use. The key observation here is that a socket requested by a process cannot already be in use unless it is by some other process within the same virtual net, and such a process is controlled by the same user.

An unusual aspect of the HOST–HOST protocol is that a process may switch its end of a connection from one socket to another. The new socket may be in any virtual net and at any HOST, and the process may initiate a switch either at the time the connection is being established, or later. The most general forms of switching entail quite complex implementation, and are not germane to the rest of this paper, so only a limited form will be explained. This limited form of switching provides only that a process may substitute one socket for another while establishing a connection. The new socket must have the same user number and HOST number, and the connection is still established to the same process. This form of switching is thus only a way of relabelling a socket, for no change in the routing of messages takes place. In the next section we document the system calls and control commands; in the section after next, we consider how login might be implemented.

SYSTEM CALLS AND CONTROL COMMANDS

Here we sketch the mechanics of establishing, switching and breaking a connection. As noted above, the NCP interacts with user processes via system calls and with other NCPs via control commands. We therefore begin with a partial description of system calls and control commands.

System calls will vary from one operating system to another, so the following description is only suggestive. We assume here that a process has several input–output paths which we will call *ports*. Each port may be connected to a sequential I/O device, and while connected, transmits information in only one direction. We further assume that the process is blocked (dismissed, slept) while transmission proceeds. The following is the list of system calls:

Init ⟨port⟩, ⟨AEN 1⟩, ⟨AEN 2⟩, ⟨foreign socket⟩

where ⟨port⟩ is part of the process issuing the Init

and ⟨AEN 1⟩, ⟨AEN 2⟩ are 8-bit AEN's (see Figure 3)

⟨foreign socket⟩ is the 40-bit socket name of the distant end of the connection.

The first AEN is used to initiate the connection; the second is used while the connection exists.

The low-order bits of ⟨AEN 1⟩ and ⟨AEN 2⟩ must agree, and these must be the complement of the low-order bit of ⟨foreign socket⟩.

The NCP concatenates ⟨AEN 1⟩ and ⟨AEN 2⟩ each with the user number of the process and the HOST number to form 40-bit sockets.

It then sends a Request for Connection (RFC) control command to the distant NCP. When the distant NCP responds positively, the connection is established and

the process is unblocked. If the distant NCP responds negatively, the local NCP unblocks the requesting process, but informs it that the system call has failed.

Listen ⟨port⟩, ⟨AEN 1⟩

where ⟨port⟩ and ⟨AEN 1⟩ are as above.

The NCP retains the ports and ⟨AEN 1⟩ and blocks the process. When an RFC control command arrives naming the local socket, the process is unblocked and notified that a foreign process is calling.

Accept ⟨AEN 2⟩

After a listen has been satisfied, the process may either refuse the call or accept it and switch it to another socket. To accept the call, the process issues the Accept system call. The NCP then sends back an RFC control command.

Close ⟨port⟩

After establishing a connection, a process issues a Close to break the connection. The Close is also issued after a Listen to refuse a call.

Transmit ⟨port⟩, ⟨addr⟩

If ⟨port⟩ is attached to a send socket, ⟨addr⟩ points to a message to be sent. This message is preceded by its length in bits.

If ⟨port⟩ is attached to a receive socket, a message is stored at ⟨addr⟩. The length of the message is stored first.

Control commands

A vocabulary of control commands has been defined for communication between Network Control Programs. Each control command consists of an 8-bit operation code to indicate its function, followed by some parameters. The number and format of parameters is fixed for each operation code. A sequence of control commands destined for a particular HOST can be packed into a single control message.

RFC ⟨my socket 1⟩, ⟨my socket 2⟩, ⟨your socket⟩, (⟨link⟩)

This command is sent because a process has executed either an Init system call or an Accept system call. A link is assigned by the prospective receiver, so it is omitted if ⟨my socket 1⟩ is a send socket.

There is distinct advantage in using the same commands both to initiate a connection (Init) and to accept a call (Accept). If the responding command were different from the initiating command, then two processes could call each other and become blocked waiting for each other to respond. With this scheme no deadlock occurs and it provides a more compact way to connect a set of processes.

CLS ⟨my socket⟩, ⟨your socket⟩

The specified connection is terminated

CEASE ⟨link⟩

When the receiving process does not consume its input as fast as it arrives, the buffer space in the receiving HOST is used to queue the waiting messages. Since only limited space is generally available, the receiving HOST may need to inhibit the sending HOST from sending any more messages over the offending connection. When the sending HOST receives this command, it may block the process generating the messages.

RESUME ⟨link⟩

This command is also sent from the receiving HOST to the sending HOST and negates a previous CEASE.

LOGGING IN

We assume that within each HOST there is always a process in execution which listens to login requests. We call this process the *logger*, and it is part of a special virtual net whose user number is zero. The logger is programmed to listen to calls on socket number 0. Upon receiving a call, the logger switches it to a higher (even) numbered socket, and returns a call to the socket numbered one less than the send socket originally calling. In this fashion, the logger can initiate 127 conversations.

To illustrate, assume a user whose identification is X'010005' (user number 5 at UCLA) signs into UCLA, starts up one of his programs, and this program wants to start a process at SRI. No process at SRI except the logger is currently willing to listen to our user, so he executes

Init, ⟨port⟩ = 1, ⟨AEN 1⟩ = 7,
⟨AEN 2⟩ = 7,
⟨foreign socket⟩ = 0.

His process is blocked, and the NCP at UCLA sends

RFC ⟨my socket 1⟩ = X'0100050107',
⟨my socket 2⟩ = X'0100050107',
⟨your socket⟩ = X'0000000200'

The logger at SRI is notified when this message is received, because it has previously executed

Listen ⟨port⟨ = 9, ⟨AEN 1⟩ = 0.

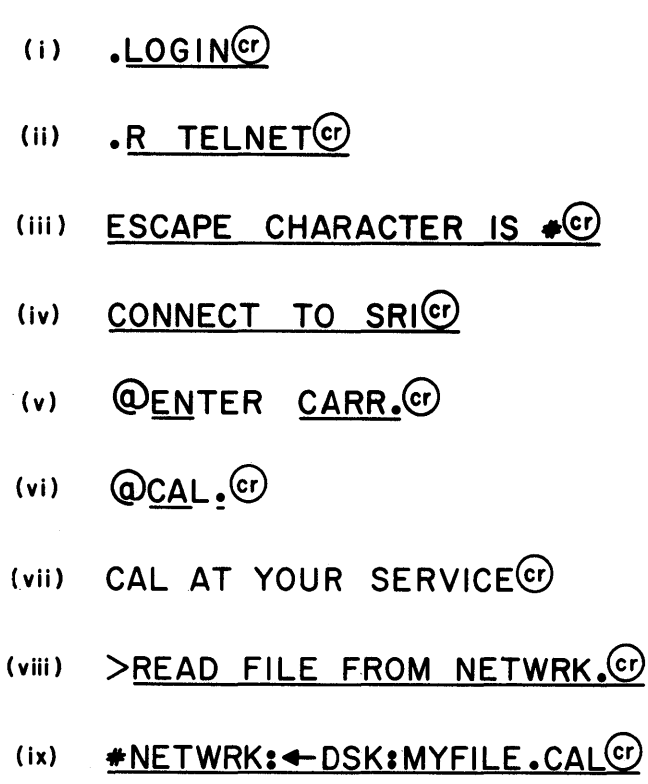


Figure 5—A typical TELNET dialog
Underlined characters are those typed by the user

The logger then executes

Accept ⟨AEN 2⟩ = 88.

In response to the Accept, the SRI NCP sends

RFC ⟨my socket 1⟩ = X'0000000200'
⟨my socket 2⟩ = X'0000000258'
⟨your socket⟩ = X'0100050107'
⟨link⟩ = 37

where the link has been chosen from the set of available links. The SRI logger then executes

Init ⟨port⟩ = 10
⟨AEN 1⟩ = 89, ⟨AEN 2⟩ = 89,
⟨foreign socket⟩ = X'0100050106'

which causes the NCP to send

RFC ⟨my socket 1⟩ = X'0000000259'
⟨my socket 2⟩ = X'0000000259'
⟨your socket⟩ = X'0100050106'

The process at UCLA is unblocked and notified of the successful Init. Because the SRI logger always initiates a connection to the AEN one less than it has just been connected to, the UCLA process then executes

Listen ⟨port⟩ = 11
⟨AEN 1⟩ = 6

and when unblocked,

Accept ⟨AEN 2⟩ = 6.

When these transactions are complete, the UCLA process is doubly connected to the logger at SRI. The logger will then interrogate the UCLA process, and if satisfied, create a new process at SRI. This new process will be tagged with the user number X'010005', and both connections will be switched to the new process. In this case, switching the connections to the new process corresponds to "passing the console down" in many time-sharing systems.

USER LEVEL SOFTWARE

At the user level, subroutines which manage data buffers and format input destined for other HOSTs are provided. It is not mandatory that the user use such subroutines, since the user has access to the network system calls in his monitor.

In addition to user programming access, it is desirable to have a subsystem program at each HOST which makes the network immediately accessible from a teletype-like device without special programming. Subsystems are commonly used system components such as text editors, compilers and interpreters. An example of a network-related subsystem is *TELNET*, which will allow users at the University of Utah to connect to Stanford Research Institute and appear as regular terminal users. It is expected that more sophisticated subsystems will be developed in time, but this basic one will render the early network immediately useful.

A user at the University of Utah (UTAH) is sitting at a teletype dialed into the University's PDP-10/50 time-sharing system. He wishes to operate the Conversational Algebraic Language (CAL) subsystem on the XDS-940 at Stanford Research Institute (SRI) in Menlo Park, California. A typical TELNET dialog is illustrated in Figure 5. The meaning of each line of dialog is discussed here.

(i) The user signs in at UTAH.
(ii) The PDP-10 run command starts up the TELNET subsystem at the user's HOST.
(iii) The user identifies a break character which causes any message following the break to be

interpreted locally rather than being sent on to the foreign HOST.

(iv) The TELNET subsystem will make the appropriate system calls to establish a pair of connections to the SRI logger. The connections will be established only if SRI accepts another foreign user.

The UTAH user is now in the pre-logged-in state at SRI. This is analogous to the standard teletype user's state after dialing into a computer and making a connection but before typing anything.

(v) The user signs in to SRI with a standard login command.

Characters typed on the user's teletype are transmitted unaltered through the PDP–10 (user HOST) and on to the 940 (serving HOST). The PDP–10 TELNET subsystem will have automatically switched to full-duplex, character-by-character transmission, since this is required by SRI's 940. Full duplex operation is allowed for by the PDP–10, though not used by most Digital Equipment Corporation subsystems.

(vi) and (vii) The 940 subsystem, CAL, is started. At this point, the user wishes to load a CAL file into the 940 CAL subsystem from the file system on his local PDP–10.

(viii) CAL is instructed to establish a connection to UTAH in order to receive the file. "NETWRK" is a predefined 940 name similar in nature to "PAPER TAPE" or "TELETYPE".

(ix) Finally, the user types the break character (#) followed by a command to his PDP–10 TELNET program, which sends the desired file to SRI from Utah on the connection just established for this purpose. The user's next statement is in CAL again.

The TELNET subsystem coding should be minimal for it is essentially a shell program built over the network system calls. It effectively establishes a shunt in the user HOST between the remote user and a distant serving HOST.

Given the basic system primitives, the TELNET subsystem at the user HOST and a manual for the serving HOST, the network can be profitably employed by remote users today.

HIGHER LEVEL PROTOCOL

The network poses special problems where a high degree of interaction is required between the user and a particular subsystem on a foreign HOST. These problems arise due to heterogeneous consoles, local operating system overhead, and network transmission delays. Unless we use special strategies it may be difficult or even impossible for a distant user to make use of the more sophisticated subsystems offered. While these difficulties are especially severe in the area of graphics, problems may arise even for teletype interaction. For example, suppose that a foreign subsystem is designed for teletype consoles connected by telephone, and then this subsystem becomes available to network users. This subsystem might have the following characteristics.

1. Except for echoing and correction of mistyping, no action is taken until a carriage return is typed.
2. All characters except " ↑ ", "←" and carriage return are echoed as the character typed.
3. ← causes deletion of the immediately preceding accepted character, and is echoed as that character.
4. ↑ causes all previously typed characters to be ignored. A carriage return and line feed are echoed.
5. A carriage return is echoed as a carriage return followed by a line feed.

If each character typed is sent in its own message, then the characters

H E L L O ← ← P c.r.

cause nine messages in each direction. Furthermore, each character is handled by a user level program in the local HOST before being sent to the foreign HOST.

Now it is clear that if this particular example were important, we would quickly implement rules 1 to 5 in a local HOST program and send only complete lines to the foreign HOST. If the foreign HOST program could not be modified so as to not generate echoes, then the local program could not only echo properly, it could also throw away the later echoes from the foreign HOST. However, the problem is not any particular interaction scheme; the problem is that we expect many of these kinds of schemes to occur. We have not found any general solutions to these problems, but some observations and conjectures may lead the way.

With respect to heterogeneous consoles, we note that although consoles are rarely compatible, many are equivalent. It is probably reasonable to treat a model 37 teletype as the equivalent of an IBM 2741. Similarly, most storage scopes will form an equivalence class, and most refresh display scopes will form another. Furthermore, a hierarchy might emerge with members of one class usable in place of those in another, but not vice versa. We can imagine that any scope might be an adequate substitute for a teletype, but hardly the reverse. This observation leads us to wonder if a network-wide language for consoles might be possible. Such a language would provide for distinct treatment of different classes of consoles, with semantics ap-

propriate to each class. Each site could then write interface programs for its consoles to make them look like network standard devices.

Another observation is that a user evaluates an interactive system by comparing the speed of the system's responses with his own expectations. Sometimes a user feels that he has made only a minor request, so the response should be immediate; at other times he feels he has made a substantial request, and is therefore willing to wait for the response. Some interactive subsystems are especially pleasant to use because a great deal of work has gone into tailoring the responses to the user's expectations In the network, however, a local user level process intervenes between a local console and a foreign subsystem, and we may expect the response time for minor requests to degrade. Now it may happen that all of this tailoring of the interaction is fairly independent of the portion of the subsystem which does the heavy computing or I/O. In such a case, it may be possible to separate a subsystem into two sections. One section would be the "substantive" portion; the other would be a "front end" which formats output to the user, accepts his inputs, and controls computationally simple responses such as echoes. In the example above, the program to accumulate a line and generate echoes would be the front end of some subsystem. We now take notice of the fact that the local HOSTs have substantial computational power, but our current designs make use of the local HOST only as a data concentrator. This is somewhat ironic, for the local HOST is not only poorly utilized as a data concentrator, it also degrades performance because of the delays it introduces.

These arguments have led us to consider the possibility of a Network Interface Language (NIL) which would be a network-wide language for writing the front end of interactive subsystems. This language would have the feature that subprograms communicate through network-like connections. The strategy is then to transport the source code for the front end of a subsystem to the local HOST, where it would be compiled and executed.

During preliminary discussions we have agreed that NIL should have at least the following semantic properties not generally found in languages.

1. Concurrency. Because messages arrive asynchronously on different connections, and because user input is not synchronized with subsystem output, NIL must include semantics to accurately model the possible concurrencies.
2. Program Concatenation. It is very useful to be able to insert a program in between two other programs. To achieve this, the interconnection of programs would be specified at run time and would not be implicit in the source code.
3. Device substitutability. It is usual to define languages so that one device may be substituted for another. The requirement here is that any device can be modeled by a NIL program. For example, if a network standard display controller manipulates tree-structures according to messages sent to it then these structures must be easily implementable in NIL.

NIL has not been fully specified, and reservations have been expressed about its usefulness. These reservations hinge upon our conjecture that it is possible to divide an interactive subsystem into a transportable front end which satisfies a user's expectations at low cost and a more substantial stay-at-home section. If our conjecture is false, then NIL will not be useful; otherwise it seems worth pursuing. Testing of this conjecture and further development of NIL will take priority after low level HOST–HOST protocol has stabilized.

HOST/IMP INTERFACING

The hardware and software interfaces between HOST and IMP is an area of particular concern to the HOST organizations. Considering the diversity of HOST computers to which a standard IMP must connect, the hardware interface was made bit serial and full-duplex. Each HOST organization implements its half of this very simple interface.

The software interface is equally simple and consists of messages passed back and forth between the IMP and HOST programs. Special error and signal messages are defined as well as messages containing normal data. Messages waiting in queues in either machine are sent at the pleasure of the machine in which they reside with no concern for the needs of the other computer.

The effect of the present software interface is the needless rebuffering of all messages in the HOST in addition to the buffering in the IMP. The messages have no particular order other than arrival times at the IMP. The Network Control Program at one HOST (e.g., Utah) needs waiting RFNM's before all other messages. At another site (e.g., SRI), the NCP could benefit by receiving messages for the user who is next to be run.

What is needed is coding representing the specific needs of the HOST on both sides of the interface to make intelligent decisions about what to transmit next over the channel. With the present software interface, the channel in one direction once committed to a particular message is then locked up for up to 80 milli-

seconds. This approaches one teletype character time and needlessly limits full-duplex, character by character, interaction over the net. At the very least, the IMP/HOST protocol should be expanded to permit each side to assist the other in scheduling messages over the channels.

CONCLUSIONS

At this time (February 1970) the initial network fo four sites is just beginning to be utilized. The communications system of four IMPs and wide band telephone lines have been operational for two months. Programmers at UCLA have signed on as users of the SRI 940. More significantly, one of the authors (S. Carr) living in Palo Alto uses the Salt Lake PDP–10 on a daily basis by first connecting to SRI. We thus have first hand experience that remote interaction is possible and is highly effective.

Work on the ARPA network has generated new areas of interest. NIL is one example, and interprocess communication is another. Interprocess communication over the network is a subcase of general interprocess communication in a multiprogrammed environment. The mechanism of connections seems to be new, and we believe this mechanism is useful even when the processes are within the same computer.

REFERENCES

1 L ROBERTS
The ARPA network
Invitational Workshop on Networks of Computers
Proceedings National Security Agency p 115 ff 1968

2 R M RUTLEDGE et al
An interactive network of time-sharing computers
Proceedings of the 24th National Conference Association for Computing Machinery p 431 ff 1969

3 F E HEART R E KAHN S M ORNSTEIN W R CROWTHER D C WALDEN
The interface message processor for the ARPA computer network
These Proceedings

A comparative study of management decision-making from computer-terminals

by C. H. JONES

Harvard University
Cambridge, Massachusetts

and

J. L. HUGHES and K. J. ENGVOLD

IBM Corporation
Poughkeepsie, New York

INTRODUCTION

The advent of interactive computer systems and cathode ray tube terminals promises great achievements in the area of managerial decision-making. Thus far, however, graphic terminals and light pens have been used mostly to solve scientific, engineering, and mathematical problems. They have been applied relatively little to the types of problems frequently encountered by business management. As a result, objective data on their effectiveness in this environment are largely lacking. In order to provide some data, a study was undertaken to observe how the managerial decision-making process might be improved by a graphic man-computer interface and to measure quantitatively the gains that might be achieved.

Many resource allocation tasks faced by management are combinatorial. Capital expenditure budgets, personnel assignment, project selection, and many production scheduling tasks all have this feature in common. Demonstrating that a computer can be helpful in solving one managerial problem of this kind would therefore suggest its extrapolation to other such problems. Since job shop scheduling has frequently been considered to be the prototype of many complex combinational problems faced by industry, it was chosen as the problem for investigation in this study.[1,4,8,9]

A great deal of effort has been devoted to finding the algorithmic and heuristic solutions to selecting the best sequence for processing jobs through the different machines in a job shop. At present, there is still no feasible optimizing procedure for selecting the best sequence from among the astronomical number of sequences possible even in a medium-sized job shop. In this sense, schedule-making is still an art.[13]

Since no single technique can satisfactorily provide an optimal answer, some recent effort[2,3,5,7,10,11] has been based on the premise that a production scheduler would find it helpful to have a computer assist him in generating and evaluating a number of alternative schedules. The scheduler could then bring to bear human cognition in further improving these computer-generated schedules and in selecting the best schedule based on his current knowledge of priority, cost, and personnel factors.

In order to study the effect of different computer interfaces on the ability of managers to generate profitable schedules using the symbiotic model described, both a typewriter and a cathode-ray-tube terminal were programmed to control the job shop scheduling model. Because its light pen allowed greater ease and speed of operation, the display terminal promised to provide more effective man-machine interaction than the typewriter terminal. A study was therefore designed to test two hypotheses: (1) that computer-aided job scheduling using either terminal was superior to manual job scheduling, and (2) that a display terminal was superior to a typewriter terminal for job scheduling.

DESCRIPTION OF JOB SHOP MODEL

The shop contains six machines: a lathe, a grinder, two boring machines, and two heat treating furnaces. The scheduler must devise a three-day job schedule for nine jobs presented to him for acceptance or rejection. Each job carries with it a selling price, delivery date, penalty for lateness, fixed sequence of two to five operations on the various machines, and amount of time for each operation. The boring operation includes three different set-ups with varying time requirements for changing set-ups. The scheduler can authorize up to eight hours of overtime on each machine each day.

Although in real life a production planner has to concern himself with a complex goal which includes terms relating to customer service, worker satisfaction, machine efficiency, etc., the scheduler in this model is asked only to maximize profits. The calculation of the profit includes the penalty costs of late deliveries and overtime premium costs. The simplification of the goal permitted the results of schedules produced by different methods to be compared objectively. Three different methods were studied: manual, typewriter terminal, and display terminal.

MANUAL JOB SCHEDULING

For several years the job shop problem has been given at the Harvard Business School to graduate students and business executives to be solved with pencil and paper. Most schedulers have followed a similar procedure, consisting essentially of the following five steps:

1. They make general arithmetic calculations based on noting the urgent jobs, profitable jobs, and loads on different machines.
2. They lay out some form of Gantt chart or paper analogue of the shop, usually consisting of a row for each machine and a column for each hour.
3. They fill in blocks of time for job operations on different machines. They are rarely able to verbalize the process by which they decide on which operation to put in next. Their main goal is to lay out some feasible way of getting through the three-day period.
4. They "fine-tune" this schedule by such adjustments as adding a little overtime (*e.g.*, "Job 2 is three hours late. If I work the lathe overtime on Day 1, I can get it out on time."), or swapping two jobs (*e.g.*, "I need to get some work to the furnace earlier, so I'll put Job 4 on the lathe ahead of Job 6 in order to use the furnace on the second day."). This fine-tuning can greatly improve a schedule.
5. They calculate the costs and profits resulting from the final schedule.

There is some looping and cycling through certain of the steps, but the five steps are usually followed in this sequence. The value of the final schedule achieved appears to depend largely upon two factors—the characteristics of the starting schedule and the improvements that can be made in that schedule.

Obviously, not all first pass schedules are equally good bases for making improvements. One may involve higher costs than another. One will highlight a critical swap which will make it easy to achieve a larger increase in profit, while another will hide the opportunity for such a swap in a way that requires the patience and skill of a cryptographer to uncover it.

The task of laying out a first pass schedule is quite time consuming. It takes the planner approximately forty-five minutes to an hour and a half. The result is that the manual schedulers are frequently stuck with their first pass because they do not have time to construct a second one.

COMPUTER JOB SCHEDULING

From these observations, there appeared to be four ways in which a computer could assist a planner:

1. It could perform a variety of standardized calculations giving him information on such variables as profit/hour for each job, hours of work required on critical machines, slack time on each job, etc.
2. The computer could be programmed to generate different first pass schedules based on rules selected by the scheduler. For example, a brief coded input could tell the computer "show me the schedule that would result if I accepted all jobs, assigned jobs with the earliest promise data first to empty machines, and worked enough overtime to finish the work waiting for machines at the end of each regular shift." Selecting different combinations of rules would allow scores of schedules to be generated in much less time than it takes to generate one manually. The decision maker could then select the best of these for fine-tuning.
3. The computer could present the information in a Gantt chart format to help the decision maker to scan it and to make improvements. It would be programmed to make it easy to enter modifications to the schedule.
4. The computer could carry out the bookkeeping so that the decision maker could quickly determine the profitability of his latest schedule.

TYPEWRITER TERMINAL

A Fortran computer program using an electric typewriter interface (IBM 1050 or 2741) to provide these capabilities for job scheduling has been described elsewhere.[7] The typewriter terminal, however, presents several obstacles to an effective "conversation" between the decision maker and the computer which a graphic terminal promises to remove:

1. Printout on the electric typewriter is slow. The time lag involved in printing a 50-line schedule at about 4 seconds a line is noticeable and breaks the concentration of the decision maker. A cathode ray tube can output an entire page of data simultaneously.
2. An electric typewriter is not well suited to making small changes in a schedule because a complete new printout is required each time. On a cathode ray tube, it is possible to change only a few numbers on a page without disturbing the rest.
3. It is cumbersome to type in code identifying the specific job, operation, and machine involved in a proposed change. With a light pen, the same input can be made merely by pointing at a location on a Gantt chart. Choosing a rule with a light pen from a table of alternatives also produces fewer errors than typing coded input, particularly for an inexperienced typist.
5. A cathode ray tube allows the presentation of graphic displays which may enhance the decision maker's understanding of the problem.

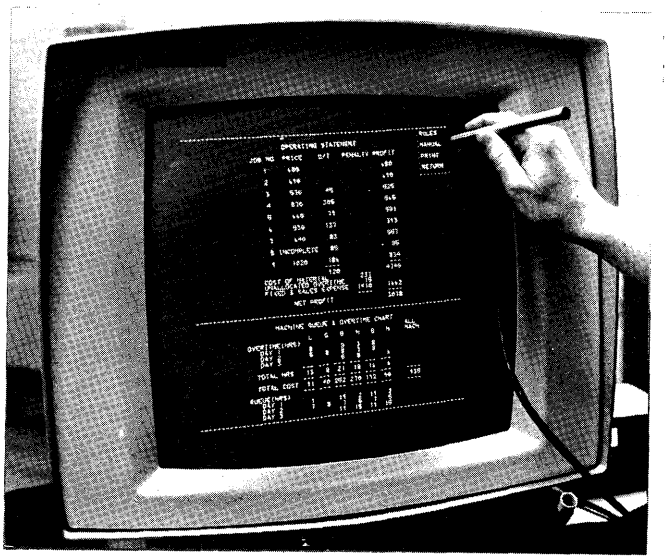


Figure 2—Operating statement for job schedule produced by selecting decision rules

GRAPHIC TERMINAL

The IBM 2250 Display Unit was therefore programmed to serve as another I/O device for the job shop scheduling model. At the display, the planner uses a light pen to select three decision rules from among three sets of rules (acceptance, sequence, overtime) displayed on the screen (Figure 1). He then light-pens "RUN" in order to view an operating

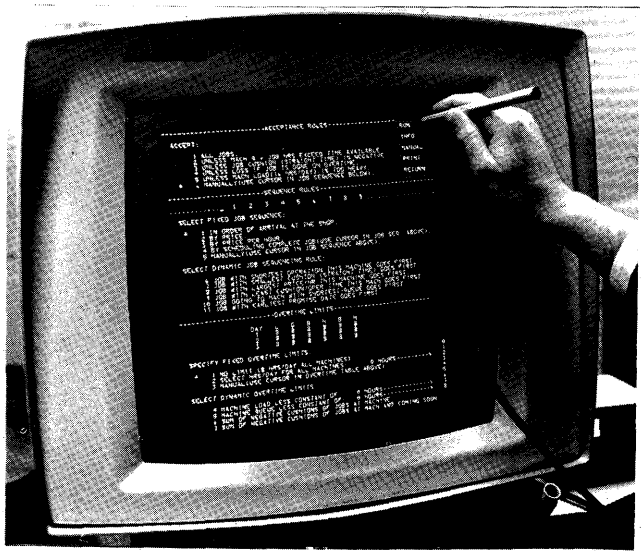


Figure 1—Decision rules for job shop scheduler, selected by light pen at IBM 2250 display unit

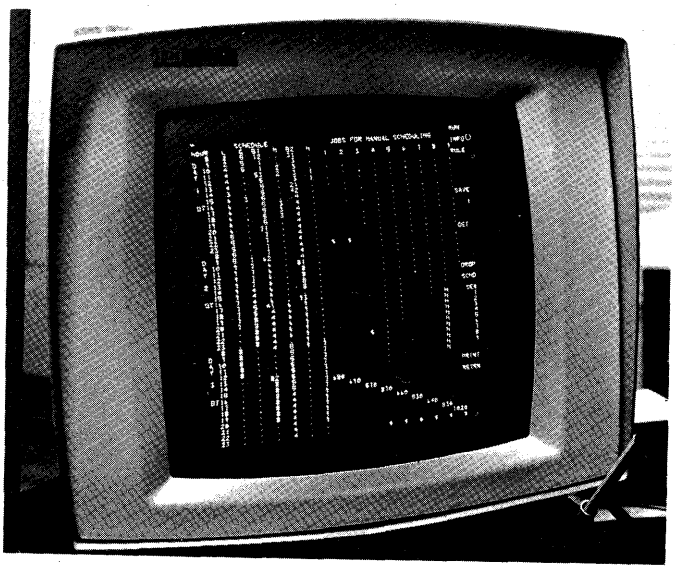


Figure 3—Job schedule produced by selecting decision rules

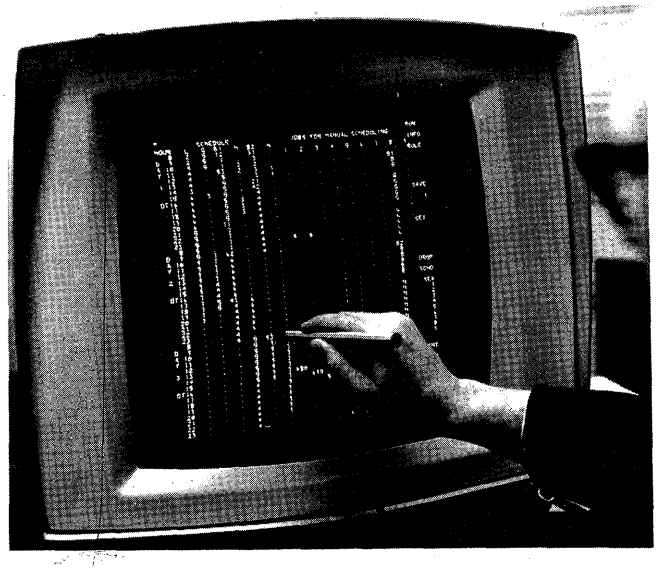


Figure 4—Job schedule, produced by selecting decision rules, being adjusted with the light pen

statement showing the net profit or loss from scheduling, errors in scheduling, and a summary of machine queue and overtime hours (Figure 2). By light-penning "MANUAL", he can examine the detailed hourly job schedule generated by the rules (Figure 3) and rearrange the job operations in the schedule manually with the light pen in an attempt to improve it. By means of the light pen, operations may be split, moved up or down the job schedule, or moved back and forth between the hourly job schedule on the left and the jobs-to-be-scheduled section at the right. The scheduler moves operations by first light-penning the operation to be moved and then the location to which the operation is to be moved (Figure 4).

Instead of selecting decision rules and generating a schedule automatically, the planner can fill in the entire schedule manually by transferring job operations from the list of jobs to be scheduled on the right to the job schedule on the left (Figure 5). This operation is not exercised frequently, however, because generating a schedule by selecting rules is more efficient. The planner can also obtain printed copy of any display by light-penning "PRINT." In addition, he can store highly profitable schedules by light-penning "SAVE" and retrieve them later by light-penning "GET."

The program for the display unit also provides additional information to the planner to aid him in making acceptance and sequencing decisions (Figure 6). If he light-pens the variable in the heading of any column (PRICE, PRICE PER HOUR, etc.) under the Sequence-of-Jobs section at the bottom, the display automatically puts the nine jobs in the order called for by the variable. For example, light-penning "PRICE" lists the jobs in descending order by price. When a satisfactory job sequence is obtained, the sequence and acceptance rules selected are transferred back to the original rules panel (Figure 1) by light-penning "RULES," after which the simulation is run from the rules panel.

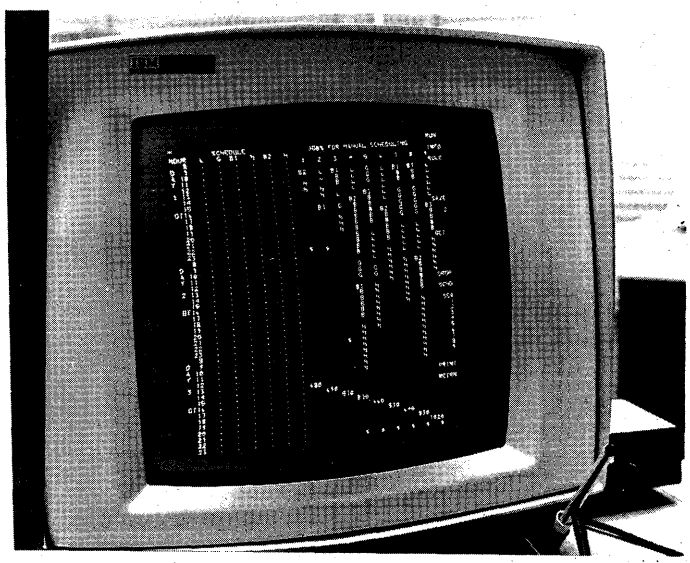


Figure 5—Empty job schedule and list of jobs to be scheduled manually by light pen

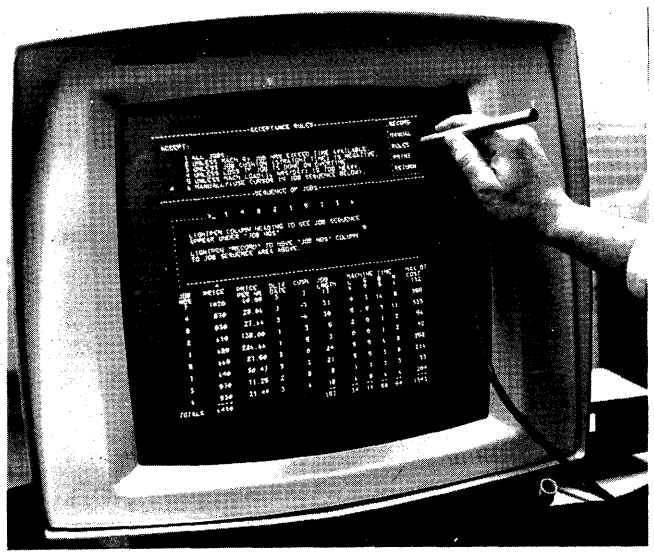


Figure 6—Supplementary information panel to assist scheduler in making acceptance and sequencing decisions

TABLE I—Performance Data on Preliminary & Maximum Profit Schedules for Manual, Typewriter, and Display Samples

	Preliminary Schedule			Maximum Profit Schedule				
	Profit	Jobs Omitted	Errors	Profit	Jobs Omitted	Errors	No. Runs	Increase In Profit
Manual Sample (n = 5 teams)								
	$ 914	6	0	$2923	2	0	1	$2009
	2546	3	0	3555	1	0	1	1009
	1411	4	3	1880	3	2	2	469
	1913	3	4	2838	2	1	1	925
	3425	1	1	3425	1	1	3	0
M	2042	3.4	1.6	2924	1.8	.8	1.6	882
SD	982	1.8	1.8	661	.8	.7	.9	748
Typewriter Sample (n = 5 teams)								
	2739	2	3	3492	1	0	18	753
	1662	5	0	3402	1	0	11	1740
	1365	5	0	3510	1	0	10	2145
	2714	3	1	3573	1	0	10	859
	2475	3	4	3188	1	0	11	713
M	2191	3.6	1.6	3433	1	0	12	1242
SD	636	1.3	1.8	150	0	0	3.4	657
Display Sample (n = 6 teams)								
	3354	1	0	3444	1	0	26	90
	3448	1	2	3448	1	0	14	0
	695	4	0	3444	1	0	23	2749
	2694	2	4	3576	1	0	51	882
	3521	1	2	3501	1	0	22	− 20
	2737	3	1	3477	*	0	15	740
M	2741	2	1.5	3466	1	0	25	724
SD	1065	1.3	1.5	52	0	0	13.5	1058

* Data missing.

THE EXPERIMENT

Thirty-two production managers and schedulers from thirteen companies in the neighborhood of the IBM Education Center in Poughkeepsie, New York, participated in the study. They were formed into 16 two-man teams and assigned by a randomization procedure (slightly modified by the exigencies of human and computer availability) to one of the three scheduling techniques: manual, typewriter terminal and graphic terminal.

The typewriter terminal was connected through public telephone lines to the MIT time-sharing computer (an IBM 7094) in Cambridge, Massachusetts, while the graphic terminal was directly connected to an IBM 360/40 in the IBM Education Center in Poughkeepsie. In the latter, the scheduling program was catalogued as a job under OS/360 and controlled by a display processing and tutorial system developed at the Poughkeepsie Education Center.[6] On the MIT computer, the program was written in Fortran and operated under MIT's CTSS. The only difference between the programs stemmed from the special capabilities and limitations of the input-output devices.

Each team received a 55-minute introductory session in which they were presented with a description of the scheduling problem and asked to devise manually a three-day schedule for the hypothetical job shop. After a five-minute break, each team was sent to its assigned experimental method (typewriter or display terminal) to see if it could devise a better schedule. During these one hour and twenty-five minute experimental sessions, the authors explained how to use the terminal and remained present to answer questions. Instructions and descriptions of the scheduling rules were also provided in printed form. After the experimental session, the manual participants were given a chance to try out the computer terminals. In addition, the type-

writer teams were given an opportunity to work at the display terminals.

The most profitable schedules achieved by each team during the introductory and experimental periods were collected for analysis. A program was written to collect data on various aspects of each team's performance at the terminals, such as amount of profit, number of schedules generated, number of jobs scheduled, and number of scheduling errors. In addition, the participants responded to written questionnaires asking about their previous experience and their attitude toward the scheduling task and the tools available to them.

FINDINGS

Table I summarizes the performance data for the schedules produced by the manual, typewriter, and display samples. The differences among the three samples in mean profit for the preliminary and maximum profit schedules were not significant by analysis of variance (ANOVA). The mean increase in profit from preliminary to maximum schedule was also not significant. However, Bartlett's test of homogeneity of variance for the maximum schedule profits was significant at the .01 level. The display group had a standard deviation of only $52, the typewriter group $150, and the manual group $661 (Table 1). All but one of the eleven display and typewriter teams had a maximum profit above $3,400, but three of the five manual teams did not achieve this figure. Thus, one result of computer scheduling apparently was to help the weaker teams—particularly those in the display sample—to raise their maximum profits closer to those of the top schedulers. On the other hand, there was little difference in profits among the top schedulers in all three samples.

On other performance variables (Table I), the expected differences in favor of the computer teams appeared. On the maximum profit schedule, each computer team omitted only one job, while three manual teams failed to schedule two or three jobs. The differences in mean jobs omitted among the three samples were significant at the .05 level by ANOVA. Three of the manual teams also made one or two errors in scheduling, while the computer program prevented the typewriter and display teams from making any errors.

Striking differences among the three samples occurred in the number of runs or job schedules generated. The display sample had a mean of 25 schedules, the typewriter sample 12 schedules, and the manual sample 1.6 schedules. The corresponding standard deviations were 13.5, 3.4, and .9. The differences in means and variances were significant at the .01 level by ANOVA and Bartlett's test, respectively. The typewriter and display teams thus not only generated more schedule runs, but showed more variation in the number generated. The size of the difference between the display and typewriter teams indicated the potential advantages of the display over the typewriter in ease of selecting decision rules with a light pen and in speed of presenting results on the screen.

On the questionnaire, all the schedulers expressed positive reactions to the use of the computer terminals. Seven of the nine participants who had a chance to try both typewriter and display terminals expressed a preference for the latter.

DISCUSSION

In addition to the objective results cited above, a number of observations of the behavior of the schedulers at the terminals were made. The approaches of the typewriter and display groups varied considerably. Once a schedule was generated by the computer in response to the selection of rules, it could be improved by making minor changes, *i.e.*, "fine-tuning." The electric typewriter sample, however, found making these small adjustments too time consuming. As a result, none of them made more than one or two such changes. On the other hand, the display teams found the sliding and swapping of jobs such fun that they were seduced into a misallocation of time. They tended to spend too much time adjusting the first schedule that they generated and therefore did not have sufficient time later to adjust their best schedule. Experience with other planners has shown that they lost $50-$100 by not taking the time to adjust their final schedule. Because of this situation, the profits attained by the display sample may very well have been understated. With more time to learn to become accustomed to the display, they undoubtedly would have done better.

One of the original goals of the cathode ray tube programming was to provide an electronic analogue for constructing a two-dimensional job schedule by means of pencil and paper. Despite the ease of light-penning job operations into a blank schedule displayed on the screen, the display teams all preferred to use rule combinations to build a starting schedule rather than fitting jobs into the schedule one by one.

If the three approaches to scheduling the shop are treated as a continuous spectrum running from no communication with a fast calculating device (the manual groups) through mediocre communication (the typewriter groups) to easy communication (the display groups), there is an interesting shift in the problem-solving techniques of schedulers in each sample.

The manual teams were clearly trying to make the best decision at each decision point. Thus, they would agonize over the decision to put Job 2 or Job 4 on the lathe. They would consider the effect of Job 2's delivery time, the amount of overtime on the lathe, the future requirements on the milling machine, etc. These decisions were usually inconclusive because it is not possible for most human minds to appreciate fully all the ramifications of the decision trees arising in job shop schedules.

The typewriter teams were less bogged down in details. Instead of arguing about a specific sequencing decision, their conversations were concerned with the merit of following decision rules which emphasized machine utilization or delivery performance. In effect they said, "We can't hope to make all the separate decisions perfectly. Let's try to figure out which decision rules should give us the best schedule." Based on existing knowledge, this is not a soluble task. Although good rationales can be given for many different rules, even small changes in the operation times and sequences can cause any rule to provide a much less satisfactory result.

The display tube teams were more pragmatic in that they spent less time discussing alternatives and more time generating schedules. They found it faster and easier to try a combination of rules than to attempt to reason out the logical value of the combination. Their replies to the attitude questionnaire showed an appreciation of the difficulty of reasoning their way to a conclusive answer. They saw the main contribution of the computer as a means of trying out more alternatives. Their colleagues using the typewriter terminals, on the other hand, emphasized the value of the computer in demonstrating the importance of using particular rules.

This study has furnished some evidence of the extent to which an interactive terminal, and particularly a display terminal, can enhance the decision-making skills of a manager engaged in solving a fairly representative kind of business problem. This approach appears to offer opportunities for improved managerial decision-making in many areas where the ability to try out many alternatives rapidly by computer and to improve the best of these alternatives by human cognition can result in more profitable decisions.

REFERENCES

1 D C CARROLL
Heuristic sequencing of single and multiple component jobs
Unpublished PhD thesis Massachusetts Institute of Technology 1966

2 D C CARROLL
Implications of on-line, real-time systems for managerial decision making
Paper prepared for presentation at the Research Conference on the Impact of New Developments in Data Processing on Management Organization and Managerial Work Sloan School of Management MIT March 29–30 1966

3 D C CARROLL
Man-machine cooperation on planning and control problems
Paper presented at the International Symposium on Long-Range Planning for Management sponsored by the International Computation Center Rome Held at UNESCO Paris September 20–24 1965

4 R W CONWAY W L MAXWELL
Network dispatching by the shortest operation discipline
Operations Research Vol 10 pp 51–73 1962

5 J C EMERY
The planning process and its formalization in computer models
Proceedings of the Second Congress of the Information System Sciences pp 369–389 1965

6 K J ENGVOLD J L HUGHES
A general-purpose display processing and tutorial system
Communications of the ACM Vol 11 pp 697–702 1968

7 R FERGUSON C H JONES
A computer-aided decision system
Management Science In press

8 W S GERE
Heuristics in job shop scheduling
Management Science Vol 13 No 3 pp 167–190 November 1966

9 B GIFFLER G L THOMPSON
Algorithms for solving production scheduling problems
Operations Research Vol 8 pp 489–503 1960

10 J C R LICKLIDER
Man-computer partnership
International Science and Technology 19ff May 1965

11 J C R LICKLIDER
Man-computer symbiosis
IRE Transactions on Human Factors in Electronics HFE–I No 1 pp 4–11 March 1960

12 H A SCHWARTZ R J HASKELL JR
A study of computer-assisted instruction in industry
Journal of Applied Psychology Vol 50 pp 360–363 1966

13 M SPITZER
The computer art of schedule-making
Datamation Vol 15 pp 84–86 1969

An interactive keyboard* for man-computer communication

by LARRY L. WEAR

Hewlett-Packard Company
Mountain View, California

and

RICHARD C. DORF

Ohio University
Athens, Ohio

INTRODUCTION

With the advent of time-sharing and remote terminals, people who have little or no programming experience are becoming computer users. One of the reasons these people have been attracted to using time-sharing computers is that the programming languages available have been made relatively simple and easy to use. BASIC is an example of the type of language that has become popular in the time-sharing community. Other languages have been developed for such fields as numerical control. Languages such as these which have been developed using terminology and syntax which are consistent with the terminology and syntax of a given field of interest are called natural languages.[1]

In the past few years there has been a considerable amount of discussion about the use of natural languages for man-computer systems.[2] Although many feel that natural languages are the wave of the future,[3] some believe that there are some problems with the use of natural languages that must be solved before natural languages can become widely used. One of the problems associated with the use of a natural language is the input of statements and commands to the computer. Natural languages usually contain a number of English words and abbreviations. Because of this, statements written in a natural language often resemble sentences written in English. If an operator is a good typist, entering a natural language statement with a teletype or similar device presents no problem. However, if the operator is not a proficient typist, inputting a statement in a natural language can be a time consuming process.

In order to circumvent the problems associated with using a standard teletype, the interactive keyboard described in this paper was developed. Besides eliminating some of the problems associated with the teletype or similar device, the interactive keyboard has two other important features which enhance the performance of a man-computer communication system; they are: (1) error prevention in the form of electrical lockout of keys that would cause incorrect syntax to be generated and, (2) visual feedback to the operator in the form of lights under keys which will result in proper syntax generation. It is the visual feedback incorporated in the keyboard which makes the keyboard interactive as opposed to the static unidirectional information transfer provided by a standard keyboard. These features are described fully in the following section.

An experiment which was conducted to obtain a measure of the performance of operators using the interactive keyboard and the results of this experiment are given in a later section of this paper.

A DESCRIPTION OF AN INTERACTIVE KEYBOARD

This section contains a detailed description of the interactive keyboard that was developed by Vincent J. Nicholson, James G. Rudolph and the author at Hewlett-Packard Laboratories. An application for a patent has been filed. The keyboard has three characteristics which make it very useful as an input device for an interactive system: (1) there is no multiple

* Patent Applied For.

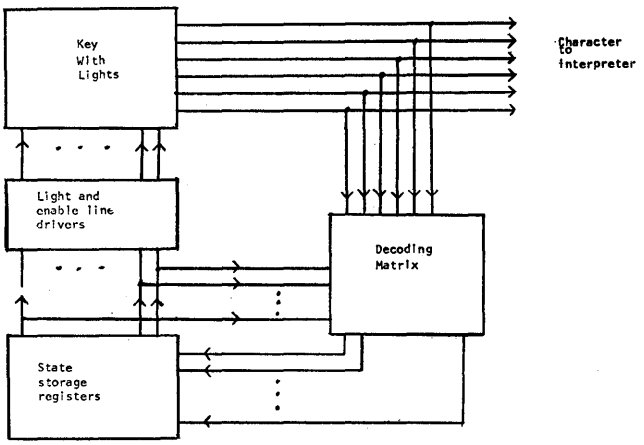


Figure 1—A block diagram of the basic elements of the interactive keyboard

use of keys, (2) lights under the keys are turned on to indicate which keys can be struck to given syntactically correct statements and (3) keys that are not lit are electrically locked out so that inputs which would result in incorrect syntax are prevented. The latter two are the characteristics which make this keyboard unique.

A block diagram that illustrates the basic elements of the keyboard is shown in Figure 1. The keyboard and its associated electronics can be broken down into four elements: the keys with lights, the decoding matrix, the state storage registers and the light and enable line drivers. The keys, which contain the lights, form the interface with the operator. When a key which has been enabled is pressed by the operator a character is sent to the processor, which in general would be the cpu of a computer; there the program resident in core will perform whatever action is required by the character. Besides going to the processor the character is transmitted to the decoding matrix. In the decoding matrix the combination of the character plus the knowledge of the present state of the keyboard is used to determine the next state of the keyboard. The state of the keyboard in this case is determined by which keys are lighted and enabled. When the next state of the keyboard has been determined the correct light and enable line drivers are activated. The process can now start over again when the next key is depressed. In the brief description given above no mention was made as to the timing required; this is because the purpose of the paper is to discuss the man-computer communication problem and not to go into a detailed analysis of hardware. For the same reason, a description of the components of each of the elements of the keyboard has been omitted.

AN INTERACTIVE KEYBOARD FOR A COMPUTER-AIDED CHECKOUT SYSTEM

The interactive keyboard described in the preceding section could be used with practically any system that required man-machine communications. In this section a description of a specific realization of such a keyboard is given. An interactive keyboard was designed for use with the Hewlett-Packard 9500A programmable checkout system.[4] A brief description of the 9500A system is given in the Appendix. The interactive keyboard was used to replace the keyboard on the teletype. The printer portion of the teletype was still used as the computer output device.

Figure 2 contains a layout of the interactive keyboard as it was designed for the 9500A system. The keys were partitioned into three general groups: words associated with standard BASIC statements, words associated with programming instruments and keys that correspond to variables, operators and miscellaneous functions such as carriage return and escape mode. These groups are enclosed by the dash lines in Figure 2.

The following example is presented to show how the keyboard functions during the input of typical statements. Suppose the operator wanted to input the following statement:

```
5 PROGRAM DVM FUNCT FREQ RANGE
  AUTO VAR A
```

If the keyboard were not initialized, that is wating for the first character of a line, the programmer would first press CR for carriage return or ESC for escape mode (these keys are always enabled and lighted). This would initialize the keyboard. When the keyboard is initalized the only keys that are lighted and enabled are the digits keys 0 through 9, CR, ESC, SPACE and the system command keys, RUN, LIST, and SCRATCH. Since it is desired to input the line shown above the

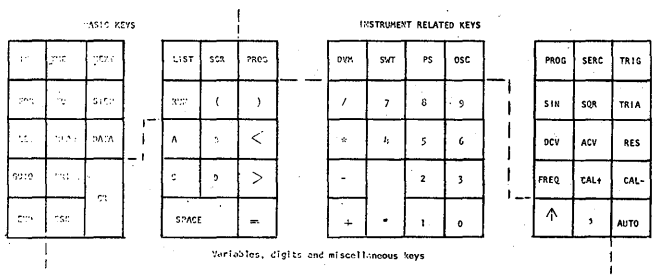


Figure 2—Layout of interactive keyboard used with the Hewlett-Packard 9500A programmable checkout system

operator would first strike the "5" key. When he had done this, a "5" would be printed on the teletype and the set of keys corresponding to permissible next inputs would be lighted and enabled. At this point the keys corresponding to legal first words at a BASIC statement will be lighted and enabled. The digits will remain lighted and enabled because it is permissible to add to the line number. The system command keys are disabled and their lights go out because if a system command is to be given it must be the first input of a line. Since CR, ESC and SPACE are always lighted and enabled they will not be for the remainder of the example.

The operator now presses the "PROG" Key. (See

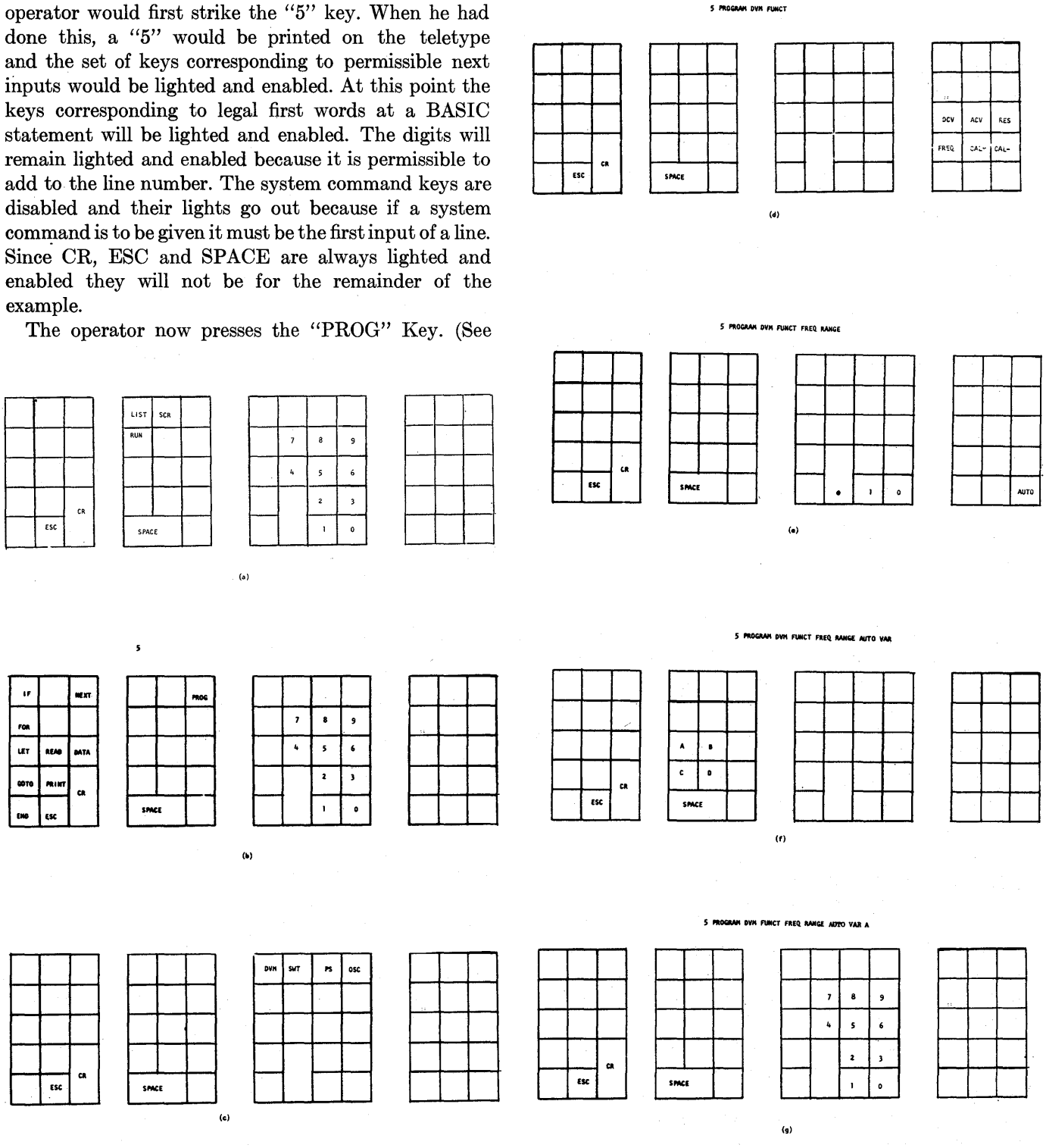


Figure 3—Illustrations (a) through (g) show which keys are illuminated and enabled during the input sequence required to enter the program statement:

5 PROGRAM DVM FUNCT FREQ RANGE AUTO VAR A

Above the keys the statement being formed is shown

reference five for a description of programming language.) This will cause "PROGRAM" to be output on the teletype and also will cause the next bank of switches to be lighted and enabled. At this point the only syntactically correct inputs are those from the keys that correspond to the programmable instruments: the "DVM," "SWT," "PS" and "OSC" keys. These keys refer to the digital volt meter, the programmable relay bank, the d.c. power supply and the oscillator, respectively. The operator now strikes the "DVM" key. When he does this "DVM FUNCT" will be typed out and the keys associated with the permissible voltmeter functions, "DCV," "ACV," "RES.' "FREQ." "CAL+" and "CAL−," will be lighted and enabled. At this point the operator has two feedback signals from the system. The abbreviation "FUNCT" has been typed out to indicate that a voltmeter function is required next and the function keys on the keyboard are lighted.

Next, the operator presses the "FREQ" key; this causes "FREQUENCY RANGE" to be typed out and the keys "I," "O," "." and "AUTO." The operator may enter a specific range such as "10000." or "10." or he may choose to use the auto ranging feature of the instrument. In this case the latter option is chosen. After he presses the "AUTO" key, "AUTO VAR" is typed out and the variable keys, "A," "B," "C" and "D," are lighted and enabled. The operator now presses the "A" key and "A" is printed and the keys "0" through "9' are lighted and enabled. The final input required is a carriage return to terminate the line and initialize the keyboard for the next line of input. Figures 3 (a) through (g) show the state of the lights for each of the steps described above and the printer output for each step.

AN EXPERIMENT USING THE INTERACTIVE KEYBOARD

An experiment was conducted on the 9500A system to measure the performance of operators using the interactive system. Four engineers who were familiar with BASIC, but who had never used the computer aided checkout system were chosen and given approximately an hour's training on the system. Two similar test procedures were given to the subjects. Two of them programmed and executed the tasks required in procedure 1 and the other two programmed and executed the tasks required in procedure 2. The time required to compose, enter and execute the programs and the number of lines required were recorded. In order to have a basis to judge the performance of the interactive keyboard, the subjects also did the required programming on a standard keyboard that had been modified for functional inputs.[5] For this experiment the two subjects who had worked with procedure 1 before were given procedure 2 and vice versa.

TABLE I—Summary of Data from Check-Out Language Experiments

	Natural		*Interactive*	
Subject	*Time*	*Lines*	*Time*	*Lines*
A	21	42	19	42
B	17	45	16	43
C	20	44	14	44
D	22	43	16	44
	80 minutes	174 lines	65 minutes	173 lines

The results of the experiment are obvious and show that the interactive keyboard improves the performance of the man-machine system significantly. A comparison between using a natural language on the teletype modified for functional input and using the interactive keyboard with natural language shows that the teletype method required 23 percent more time than the interactive keyboard method.

CONCLUSIONS

The main conclusion that can be drawn from the comparison of the data from the experiment described above is that the interactive keyboard provided a significantly improved method of man-machine communication. Two comments made by the test subjects seem worthy of noting: first, two of the subjects said that by the end of the test they were watching the keyboard almost exclusively and not referring to the teletype print-out; second, one subject said that programming with the interactive keyboard was more relaxing than programming with the teletype.

Even though the improvements in performance that were calculated above are probably not exactly correct, they do provide a positive indication that the interactive keyboard is a definite improvement over the teletype in man-machine interactive systems.

EXTENSIONS

The type of interactive terminal described above does not have to be limited to a keyboard method of entry. If one has available a CRT with light pen input the same principles can be applied to the design of the system. In this case rather than just illuminating a key with a word written on it, the word can be written on the face of the CRT and if the operator desires to

input that work he touches the CRT in the area of the word with the light pen.

With a little more imagination one could envision a system with a CRT to provide feedback to the operator and an audio processor to convert verbal commands into correct electrical signals. Since work is already being done to convert brain impulses into computer commands it is probably only a matter of time until complicated interactive systems such as these are in general use.

BIBLIOGRAPHY

1 M HALPERN
Foundations of the ease for natural-language programming
IEEE Spectrum March 1967

2 J R PIERCE
Men, machines and languages
IEEE Spectrum July 1968

3 R B MILLER
New, simple man/machine interface
EDN October 15, 1969

4 R A GRIMM
Automated testing
Hewlett-Packard Journal August 1969

5 L L WEAR
Natural languages and functional input devices for man-computer systems
Santa Clara University 1969

APPENDIX

Description of the 9500A system and a natural language for computer aided checkout

The particular configuration of the system used to conduct the experiments on is shown in Figure 4. For this experiment, the unit under test was the system itself. Measurements were made on the outputs of the 6130 programmable dc power supply and the 157 programmable oscillator and on resistors placed in the distribution unit.

The general purpose digital computer used in the system is a H-P 2114A with 8 K of core memory. Each of the instruments and input/output devices is connected to the computer through one of the 2114's interrupt connectors. The language used by the operator to communicate with the computer and the instruments is a modified version of BASIC. An earlier section of this paper contains a detailed description of the features of the modified portion of BASIC.

There are two instruments in the system that are used to supply stimuli to the unit under test; these are

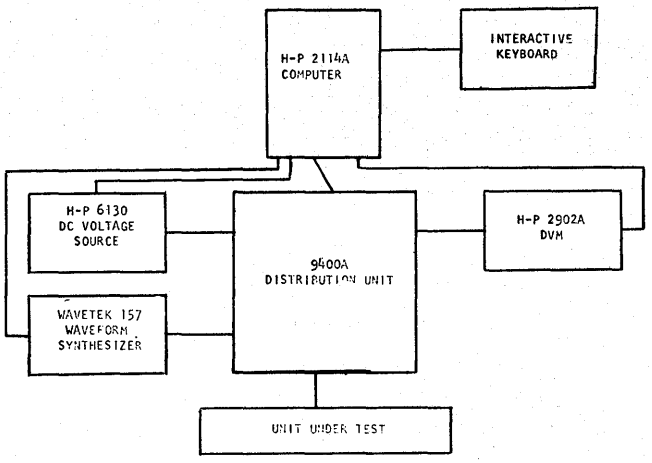


Figure 4—A block diagram of the 9500A system used to conduct the experiment on natural language and functional inputs

the Hewlett-Packard 6130 DC voltage source and the Wavetek 157 waveform synthesizer. The 6130 is capable of suppling DC voltages from −50 to +50 volts. When using this instrument, the operator must supply three parameters to the system:

1. The address of the particular 6130 to be used (in this special case there was only one 6130 so that the address was always 1).
2. The desired voltage in volts.
3. The desired current limit on the power supply.

The 157 is a programmable waveform generator. It is capable of generating sine, square and triangular waveforms with amplitudes of .001 to 10 volts peak-to-peak and with frequencies from .0001 to 1,000,000 hertz. The generator has three modes of operation program, trigger and search. In this program mode, the specified output is provided continuously from the time the instrument is programmed until a new output is requested. In the trigger mode, the output is supplied only during the time that an external trigger pulse is applied. In the search mode, the frequency of the output is determined by a dc voltage that is supplied to the 157 through a rear panel connector by external equipment. To program the waveform synthesizer the operator must, in general, supply four parameters to the system:

1. The mode of operation
2. The waveform type
3. The frequency
4. The amplitude

The 2402A DVM is the only measurement device

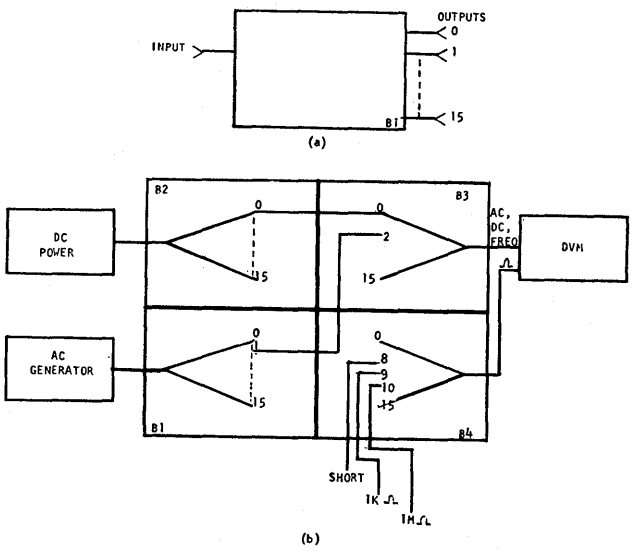


Figure 5—(a) A typical relay switch bank in the 9400A distribution (b) Interconnections between the stimuli and DVM in the 9500A demonstration system

in the system used for this experiment. The 2402A is capable of measuring DC voltages, AC voltages, frequency and DC resistance. Two other functions are available to check the calibration of the instrument; they are calibrate + and calibrate −. Besides the desired function, the operator must supply the system with two other parameters:

1. The range for the instrument
2. The identifier under which the measurement value is to be stored in the computer

The 9400A distribution unit is used to connect the supplies and measuring devices to specified pins on the 9400A. The unit contains four banks of relay operated switches. Figure 5 (a) shows a typical switch. The instrument connected to the input pin can be switched to any one of the 16 output pins. Figure 5 (b) shows how the 9400A was wired for the system used in the experiment.

To program the 9500A the operator must supply the desired output pin, 0 through 15, for each switch bank. If the operator does not want to change output connection of one of the switches, he can input a-1 to the system rather than a number between 0 and 15.

The language used with the 9500A system was a modified version of BASIC. Extensions were made to BASIC so that the system was capable of interpreting the statements used to control the various instruments. When the operator wants to give a command to one of the instruments he enters "PROGRAM" following the line number. The operator then enters the instrument, DVM, OSC, PS or SWT. Following this, modifiers associated with the instrument being used are input. For a complete description of the language see Reference 5.

Linear current division in resistive areas: Its application to computer graphics

by J. A. TURNER and G. J. RITCHIE

University of Essex
Colchester, England

INTRODUCTION

Present-day computer systems employ a variety of sophisticated peripheral equipments for the input and output of information. This paper describes a new method of obtaining (x, y) coordinate position information by means of linear current division in a resistive area.[1]

The method has applications in many fields but particularly that of Computer Graphics where it can be used as an input device in the form of a Data Tablet, as an output device in the form of a precision Cathode-Ray-Tube Display and as an alternative to the 'light-pen'.

Linear current division applied to Data Tablets differs radically from approaches used by Rand[2] and Sylvania[3] in that the operator's electronic pen or stylus injects a constant current into the tablet, the x and y coordinate position information being obtained from individual peripheral connections to the tablet. It has the advantage of being a basically accurate system which is simple to construct and therefore economical to manufacture. In addition, the coordinate information can be sampled at a high rate, $10KHz$ being achieved without difficulty on the prototype Data Tablet.

Included in this paper are the experimental results obtained from a Data Tablet—results which can be applied to any system employing the principle of linear current division. Also presented is a theoretical verification of the principle.

CURRENT DIVISION IN A RESISTIVE AREA

If a current is injected at a point in a finite resistive area, the current must emerge at the boundary of the area. The coordinate position of the current source can be described by the distribution of current magnitudes at the boundary of this area, but this distribution generally will be a very complex function of position. Establishing certain symmetrical boundary conditions yields a simple relationship between coordinate position and boundary currents, as follows.

Linear current division for the one-dimensional case

Consider a rectangular area with uniform surface resistivity as shown in Figure 1.

The edges at $x = 0$ and $x = x_0$ are reinforced with low resistivity material and are connected to earth. If a current I is injected into the area at a point (x_1, y_1), currents I_1 and I_2 flow in the conducting edges as shown.

Symmetry analysis yields the following relationship:

$$\frac{I_2}{I} = \frac{x_1}{x_0} \qquad \text{(see Appendix)}$$

This relationship has been verified experimentally using an x–y table with a constant current probe which could be moved over a rectangular sheet of commercially-available teledeltos paper, one pair of opposite edges being reinforced with a coating of high conductivity silver paint.

The results are shown in Figures 2 and 3.

Note particularly that when moving the probe in the y-direction, the current I_2 is constant whereas moving the probe in the x-direction produces a current I_2 which is linearly proportional to x position.

When making these measurements, a random error of 0.25 per cent due to variations in surface resistivity of the teledeltos paper was observed.

Linear current division for the two-dimensional case

The method for measuring probe position previously described can be extended to a two-coordinate system by presenting the resistive area with alter-

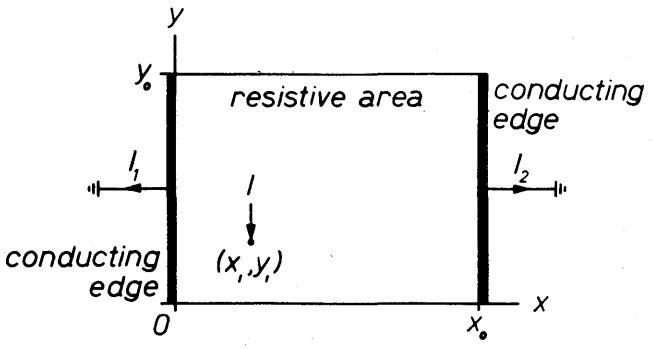


Figure 1—Current division in a resistive area

nately conducting and non-conducting opposite pairs of edges. One method by which this may be achieved is illustrated in Figure 4(a).

The periphery of the conducting area is connected to four groups of uniformly-spaced diodes, the direction of conduction of the diodes being appropriate to the polarity of the current source. In this case, the anodes of the diodes along a single edge of the resistive area are connected together and also to the emitter of an *npn* gating transistor (transistors Q_1, Q_2, Q_3 and Q_4).

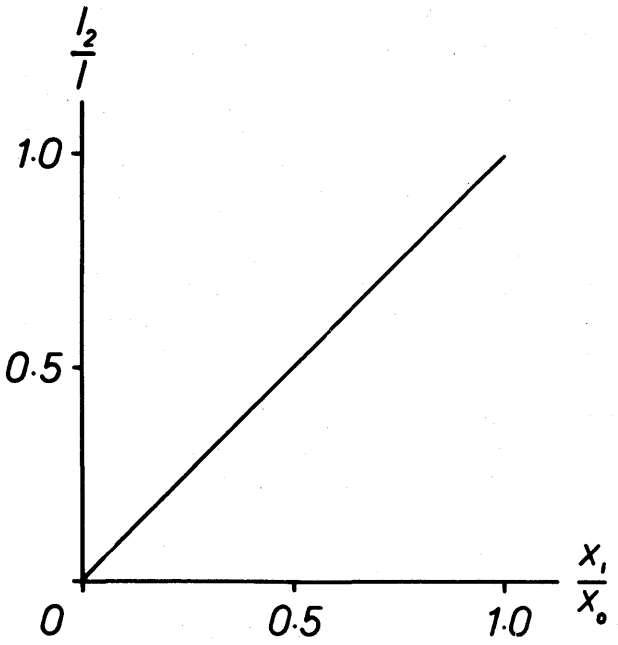


Figure 2—Linearity of current division

Consider the waveforms shown in Figure 4(b). During the 'x period,' transistors Q_2 and Q_4 are gated OFF so that $I_2 = I_4 = 0$, whereas Q_1 and Q_3 are gated ON. The diodes associated with Q_1 and Q_3 thus set up equipotential edges parallel to the y-axis so that the source current, I, splits into I_1 and I_3 giving a voltage $V_{R3} = I_3R$ which is linearly proportional to the x-coordinate position of the source. During the 'y period,' transistors Q_1 and Q_3 are OFF whereas Q_2 and Q_4 are ON; thus $V_{R2} = I_2R$ is a voltage linearly proportional to the y-coordinate position of the source.

If the gating is repetitive, integration of V_{R3} and V_{R2} respectively provide low-frequency analogue x–y

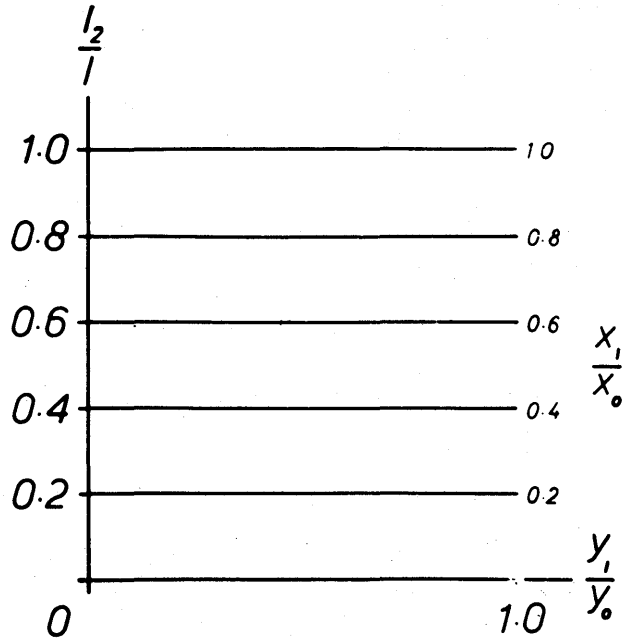


Figure 3—Constancy of current division

information; alternatively, sampling of V_{R3} and V_{R2} may be performed during the appropriate gating periods to provide instantaneous measurements.

An alternative method of operating the system is indicated in Figure 5 in which the current source is switched between $+I$ and $-I$. In this case opposite banks of diodes are arranged to conduct sequentially and are terminated in the low impedance virtual earths of operational current-to-voltage amplifiers. This method has the advantage of eliminating the varying emitter-base voltage drop associated with transistors Q_1, Q_2, Q_3 and Q_4.

NON-LINEARITIES

Although the simple one-dimensional system of Figure 1, with its completely reinforced edges, gives an ideal 'y-independent' and 'x-proportional' division of current, the two-dimensional systems described in the second section introduce non-linearities in current division. This is due to the small, discrete contact areas of the diode connections and to the fact that practical diodes have a finite and variable voltage drop when forward-biased as well as a leakage current when reverse-biased.

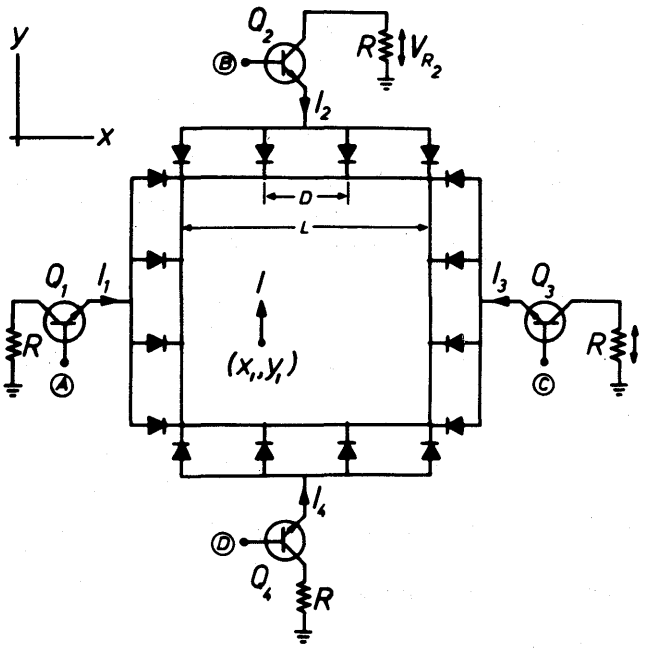


Figure 4 (a)—Basic (x, y) system

The effect of discrete contact areas

In order to separate out the effects of finite diode contact area and diode voltage drop, an experiment was performed in which a constant current probe was moved parallel to and at a fixed small distance, M, from a conducting edge. This conducting edge was provided by small circular discrete contact areas connected together using 'ideal' forward-conducting diodes, i.e., wire.

The results are shown in Figure 6.

It is seen that the current I_2 is subject to regular variations (ripple), each peak corresponding in position to a contact area. This is to be expected because the division of current between two opposite edges is

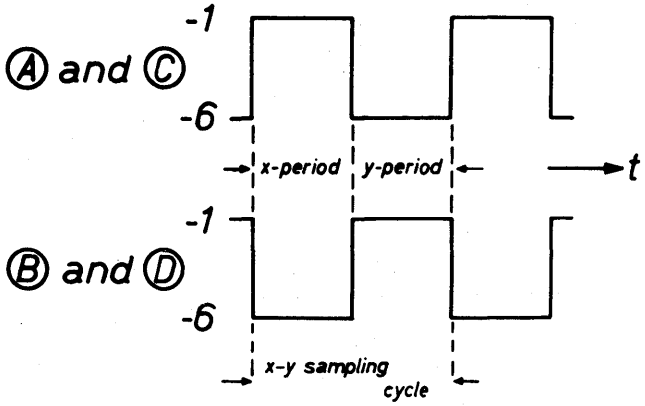


Figure 4 (b)—Voltage drive waveforms

dependent upon the relative impedances of the current paths from the current source contact point to the edges. A consequence of this behaviour is that the ripple should be reduced if the probe is traversed at a greater distance from a conducting edge or if the number of contact areas, n, on each edge is increased. These effects are illustrated in Figures 7 and 8.

Since an edge of the resistive area is required to be conducting and non-conducting during successive half-cycles of the x–y coordinate position sampling waveform, two means whereby the ripple amplitude might have been reduced were investigated. Both

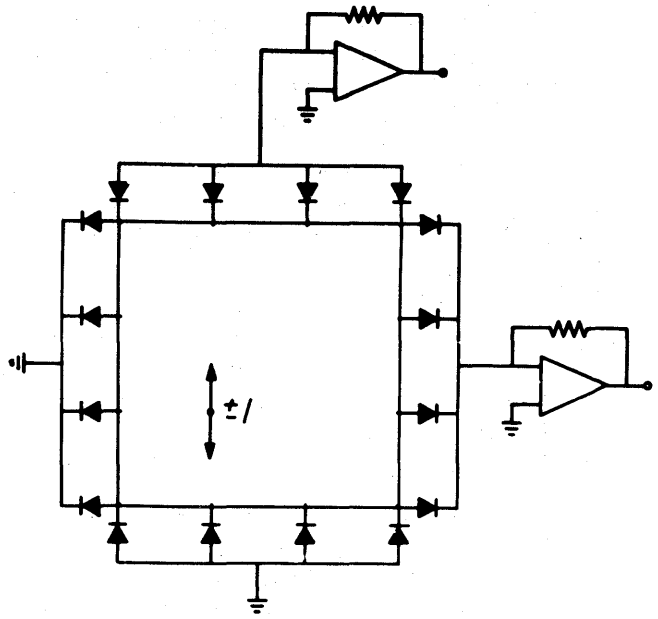


Figure 5—Alternative (x, y) system

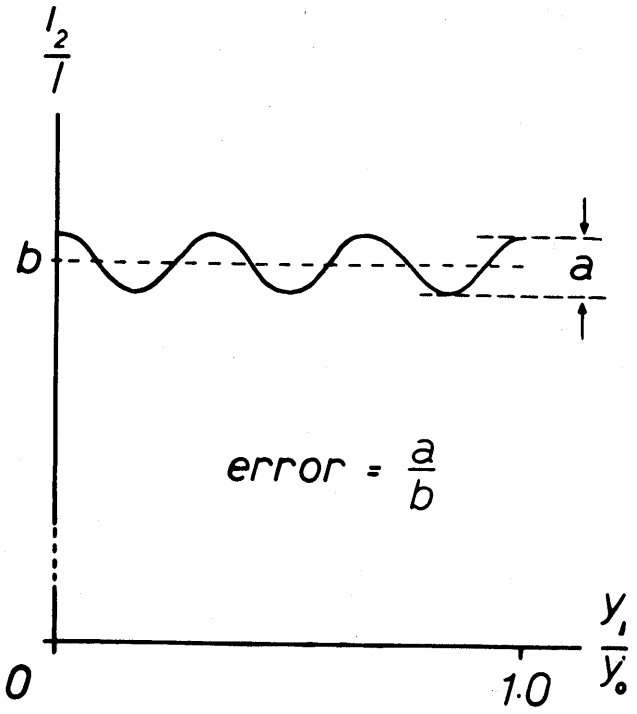


Figure 6—Ripple due to edge contacts

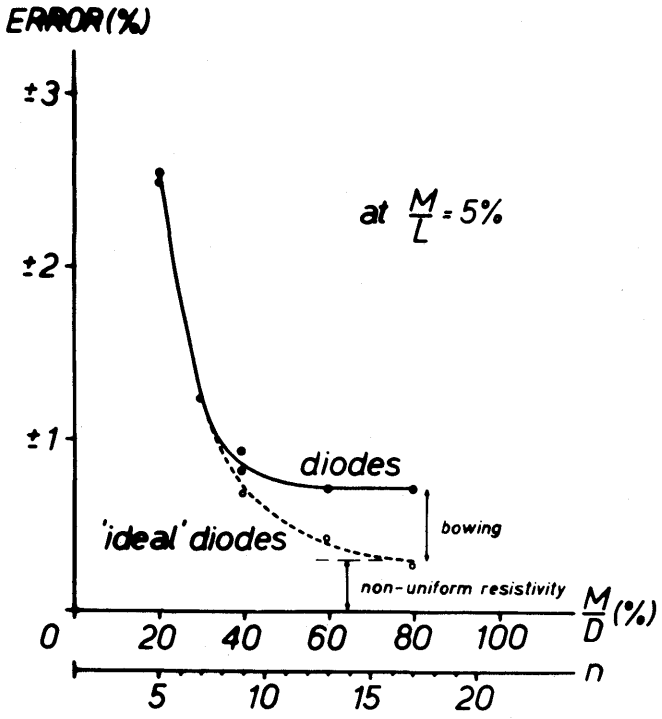


Figure 8—Variation of error with number of edge connections

attempted a closer approach to a reinforced edge than the dot contact structure described in the previous section.

The first method involved extending the dots into lines along the edge of the conducting area, measurements of ripple amplitude being taken for various contact-to-space ratios (see Figure 9).

It was found that the ripple remained substantially unchanged for contact-to-space ratios up to 75% but, more important, the current division plot showed a marked deviation from linearity in the form of stepping, even with a contact-to-space ratio as low as 25% (Figure 10). This approach was therefore rejected.

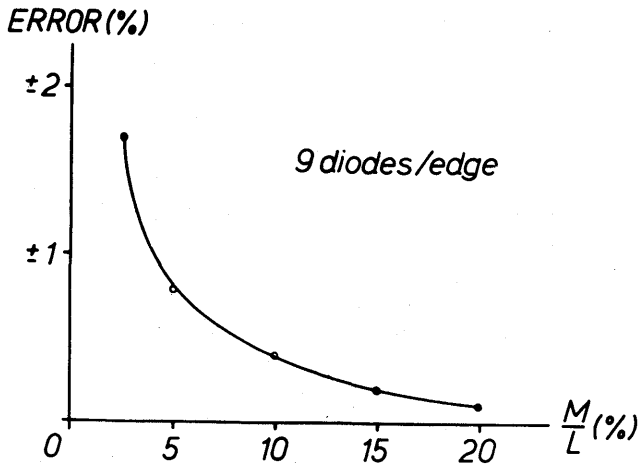


Figure 7—Variation of error with edge approach distance

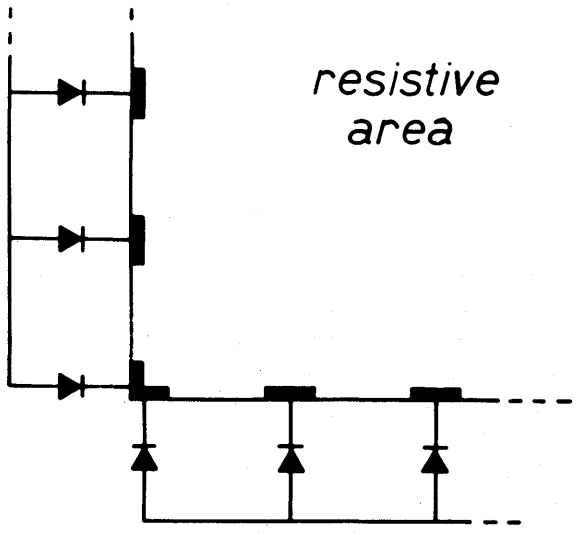


Figure 9—Extension of dot contacts into lines

The second method consisted of interposing isolated conducting dots between adjacent contact dots (Figure 11), but no change in ripple amplitude could be observed even when interposing as many as fifteen isolated dots between each pair of contact dots.

It is therefore concluded that the ripple amplitude can be reduced by:

(a) increasing the number of contact dots per edge, and
(b) increasing the minimum approach distance of the current source from an edge.

The effect of diode voltage drop (ON diodes)

Semiconductor diodes have a logarithmic current-voltage characteristic which is conveniently described by the relationship: $59mV$ change per decade of current change. Conducting diodes along a single edge in general carry currents which are different from one another; hence the edge is no longer an equipotential. This results in a 'bowing' superimposed on the ripple and gives rise to an increase in the overall error (solid curve Figure 8).

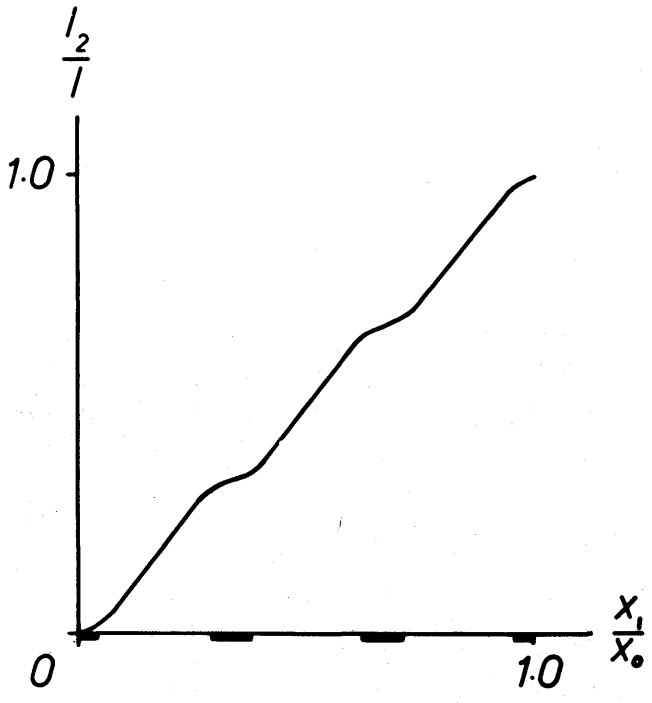


Figure 10—"Stepping" error due to finite contact dimensions

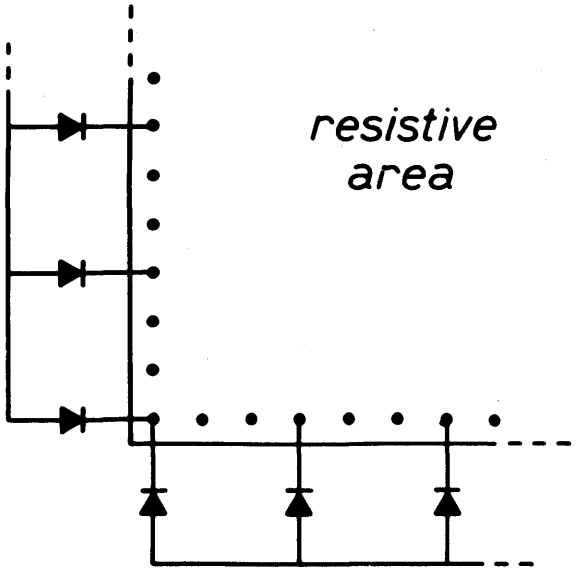


Figure 11—Interposed dot contact structure

The effect of leakage current (OFF diodes)

In order to obtain a non-conducting edge, the diodes associated with that edge are reverse-biased. Each reverse-biased diode exhibits a leakage current, and the net source current available for position measurement is therefore $(I - 2nI_s)$.

I is the source current.
n is the number of diodes per edge.
I_s is the reverse leakage current per diode.

The EC402 diode has a typical I_s as low as $0.45nA$ at 5 volts reverse-bias, yielding a total error current of $18nA$ for $n = 20$. If the source current is $1mA$, leakage will then account for a typical error of less than 0.002 per cent of full scale. This error is insignificant compared with the other factors under consideration.

Effects due to the resistive area

Non-uniform surface resistivity causes the current division between opposite edges to be non-linear. This effect can be seen in Figure 8 in which a random error of 0.25 per cent is present when using teledeltos paper. It is therefore essential to use a material with a uniform surface resistivity.

For a constant current injected into the resistive area, the voltage developed at the point of contact will depend upon the surface resistivity. Since the

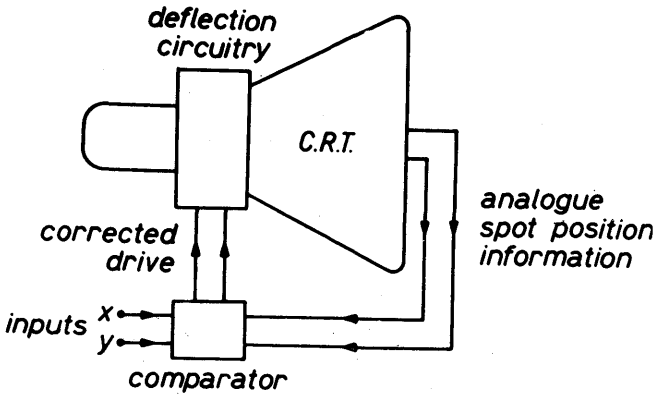


Figure 12—Feedback C.R.T. display system

'bowing' non-linearity is due to variable forward-biased diode voltages, errors due to 'bowing' can be minimised by using a material with large surface resistivity or injecting a large current. In both cases errors due to 'bowing' will be minimised when most of the voltage is dropped across the resistive area rather than the diodes.

CATHODE RAY TUBE DISPLAY SYSTEMS

The principle of linear current division can be applied most effectively in a Cathode Ray Tube in which the electron-beam is the current source and the resistive area is the aluminised backing to the phosphor, or a transparent conducting screen inner surface. The coordinates of spot position are obtained from the face of the tube allowing the tube to be included within a feedback loop, thus producing a deflection linearly proportional to input signal independent of the non-linearities within the loop (Figure 12).

A C.R.T. of this type will find applications in professional display systems such as Computer Graphic and Radar Displays.

THE DATA TABLET

The two-coordinate systems described in an earlier section may be used as a Computer Graphic Input Device or Data Tablet.

A prototype Data Tablet has been built (Figure 13) in which the analogue signals V_{R2} and V_{R3} of Figure 4(a) are connected to an x–y plotter in order to demonstrate the working capabilities of the system. x and y position information is sampled at $10KHz$ which is more than sufficiently fast for hand-written data to be reproduced without observable error.

When hand-written information is presented to the Data Tablet, variations in pressure of the stylus input can be accommodated within the voltage swing capability of the input current-source, but intermittent contact will give a false (0, 0) position. In any practical device, a finite peripheral margin will be required in order that the ripple non-linearities may be kept down to a pre-specified level. Under these conditions, a (0, 0) output will indicate that the stylus has been lifted. The lifting may be temporary in the case of intermittent contact, or semi-permanent in the case of finishing a hand-drawn line. Where intermittent contact takes place, the (0, 0) indication is obtained for a few cycles only of the (x, y) sampling waveform—in which case a peak-charging sample-and-hold circuit maintains the output at its last non-zero value. A continuous output of (0, 0) for, say, $500mS$ indicates that the stylus has been lifted.

Where digital information of position is required, the sample-and-hold circuit is replaced by an output register.

A Data Tablet in which the resistive area is transparent has the capability of being used as a 'light-pen' when the tablet is placed over the face of a Cathode-Ray-Tube. A tube of the type described in the above section with a linear x–y display is an obvious choice for this application.

SUMMARY AND CONCLUSIONS

The principle of linear current division in resistive areas has potential application in many fields, but particularly that of Computer Graphics where it may be used in both input and output devices. Attractive features of

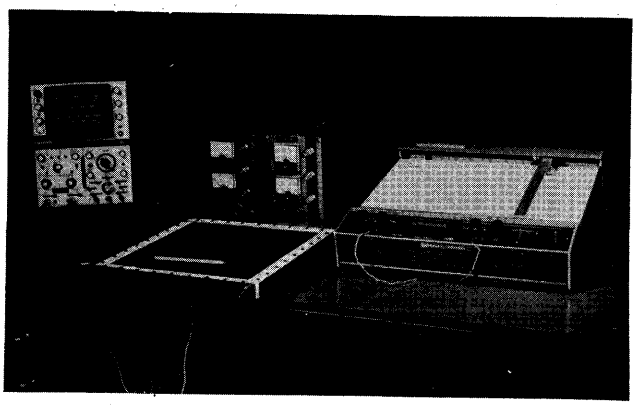


Figure 13—Prototype data tablet

the system are:

(a) High accuracy (better than 1%)
(b) Simplicity in concept and construction
(c) (x, y) sampling rate $\gg 10KHz$

The basic accuracy of the Data Tablet on which the experimental results were taken is better than 1%. With a suitable choice of margin and number of diodes per edge, the main sources of error are non-uniform surface resistivity and the effects of forward-biased diode voltages. It is expected that considerable improvement can be made by careful choice of surface resistivity and input current magnitude. At present, information which has been hand-drawn is lost to the operator, unless a graphical output system is employed. It is, however, envisaged that transparent semi-flexible resistive areas in conjunction with a pressure-sensitive hard-copy material will allow the operator to see what he has drawn at the point of contact of the stylus, in addition to obtaining hard-copy.

An area, as yet untouched in the electronics field, and in which linear current division has obvious application, is the extraction of analogue position information from a Cathode-Ray-Tube face. Output currents proportional to position are obtained when the beam current is a constant fixed value. An alternative method of operation requires a divider to evaluate I_2/I in which case a continuous indication of beam position can be obtained even when the brightness of the trace is varying. Circuitry to perform this normalised analogue division is currently under development.

ACKNOWLEDGMENTS

The authors wish to thank the Department of Electrical Engineering Science of the University of Essex, England, for the facilities provided, and Automatic Radio, Melrose, Massachusetts for development funding. G. J. Ritchie is indebted to the Science Research Council and the Marconi Company Limited for an Industrial Studentship grant.

Thanks are also due to Dr. L. F. Lind for many useful discussions regarding the appendix proof

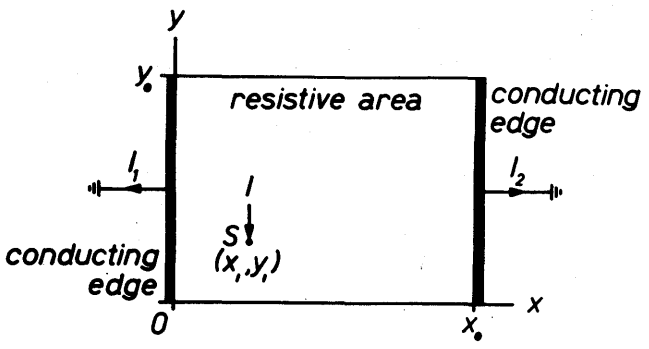


Figure 14—Current division in a resistive area

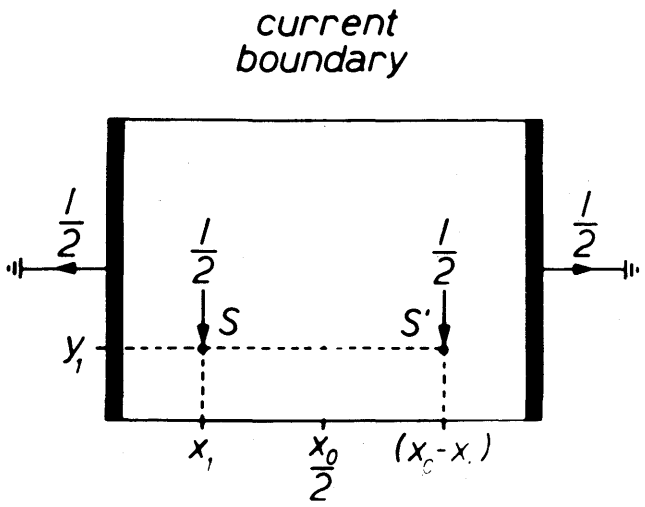


Figure 15—First-cycle even-mode configuration

REFERENCES

1 U.K. Patent Application No. 13341/69
2 T O ELLIS M R DAVIS
The Rand tablet: A man-machine communication device
Proceedings Fall Joint Computer Conference pp 325–331 1964
3 J F TEIXEIRA R P SALLEN
The Sylvania data tablet: A new approach to graphic data input
Proceedings Spring Joint Computer Conference pp 315–321 1968
4 J REED G J WHEELER
A method of analysis of symmetrical four-port networks
IRE Transactions on Microwave Theory and Techniques pp 246–252 October 1956

APPENDIX

Proof of the linearity of current division

Consider the uniform, resistive, rectangular sheet shown in Figure 14. The edges at $x = 0$ and $x = x_0$ are reinforced with a highly conductive material and are connected to earth. A current, I, is injected into the

sheet at source $S(x_1, y_1)$ causing currents I_1 and I_2 to flow in the conducting edges as shown.

It is required to establish the relationship between the currents I_1 and I_2, and the coordinate position of the source. Symmetry analysis (even and odd mode) yields the solution.[4]

Convention: A downwards arrow ($\downarrow$) denotes the injection of current, whilst an upwards arrow ($\uparrow$) denotes removal of current from the point indicated.

First Cycle: The first line of symmetry is $x = x_0/2$ with S' the image of S (Figure 15). The current, I, can be described by the sum of the even and odd mode contributions.

Even mode: Equal currents of magnitude $I/2$ injected at S and S' give, by symmetry, contributions of $I/2$ to both I_1 and I_2 irrespective of y_1 and of whether S lies in the left-hand or right-hand half-plane.

Odd mode: By symmetry $x = x_0/2$ is an equipotential at earth potential in both cases of Figure 16, but the magnitudes of the boundary currents are not known. However, these may be established by sub-dividing this first cycle odd mode into further even and odd modes. Since each half-plane of Figure 16 is bounded by two earth equipotentials, the half-plane containing S may be treated as follows.

Second Cycle: There are four possible cases for each mode.

Even mode: The axes of symmetry occur at $x = x_0/4$ or $x = 3x_0/4$ with S'' the image (Figure 17). In (a) and (b), for which $x_1 < x_0/2$, there is a contribution of $+I/4$ to I_1 and a $-I/4$ contribution to I_2.

In (c) and (d), for $x_1 > x_0/2$, there is a $+I/4$ contribution to I_2 and a $-I/4$ contribution to I_1.

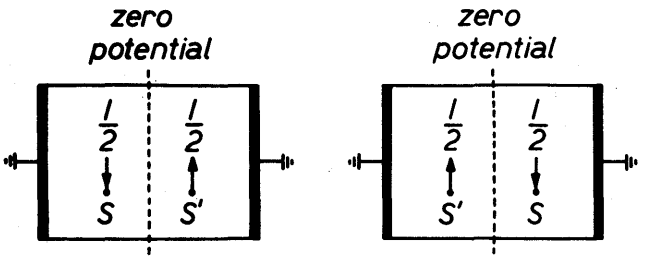


Figure 16—First-cycle odd-mode configurations

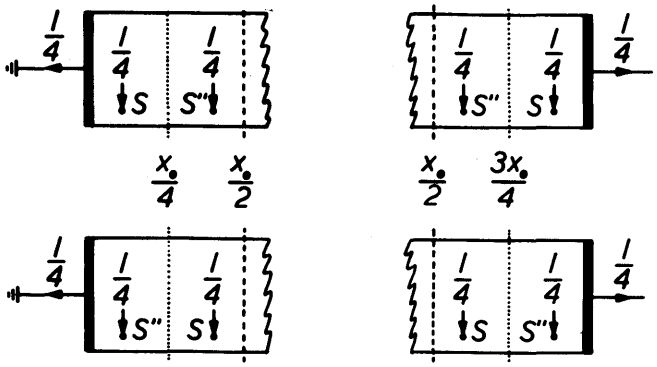


Figure 17—Second-cycle even-mode configurations

Therefore, if $x_1 < x_0/2$, i.e. $x_1/x_0 = \frac{1}{2} - \frac{1}{4} \pm$ (other terms)
then $I_2/I = \frac{1}{2} - \frac{1}{4} \pm$ (odd mode, second cycle)
and if $x_1 > x_0/2$, i.e. $x_1/x_0 = \frac{1}{2} + \frac{1}{4} \pm$ (other terms)
then $I_2/I = \frac{1}{2} + \frac{1}{4} \pm$ (odd mode, second cycle)

Study of the second cycle odd mode leads to the third cycle even mode contribution and introduces $\pm I/8$ terms in direct correspondence with the description of x_1/x_0 by the series $\frac{1}{2} + \frac{1}{4} \pm \frac{1}{8} \pm$ (other terms) or $\frac{1}{2} - \frac{1}{4} \pm \frac{1}{8} \pm$ (other terms).

The following pattern emerges:

(i) The even modes contribute to I_2/I as terms of the form $\pm\frac{1}{2}^n$ forming a series $\frac{1}{2} \pm \frac{1}{4} \pm \frac{1}{8} \ldots$, each sign being positive if the source lies in the right-hand side or negative if the source lies in the left-hand side of the appropriate sub-division of the plane.

(ii) x_1/x_0 is also described by this series with a direct correspondence of signs. (The series $\frac{1}{2} \pm \frac{1}{4} \pm \frac{1}{8} \pm \ldots \pm \frac{1}{2}^n \ldots$ is absolutely convergent, therefore this series is convergent, regardless of sign choice.)

(iii) The currents are independent of y_1 because of symmetry considerations, i.e., the images of S are at the same height as S.

$\therefore I_2/I = x_1/x_0$ and I_1/I is the complement of I_2/I

Thus, division of the current source, I, between the two conducting edges parallel to the y-axis has a linear dependence on the x-coordinate of the source and is independent of the y-position.

Remote terminal character stream processing in Multics

by J. H. SALTZER

Massachusetts Institute of Technology
Cambridge, Massachusetts

and

J. F. OSSANNA

Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

INTRODUCTION

There are a variety of considerations which are pertinent to the design of the interface between programs and typewriter-class remote terminal devices in a general-purpose time-sharing system. The conventions used for editing, converting, and reduction to canonical form of the stream of characters passing to and from remote terminals is the subject of this paper. The particular techniques used in the Multics* system are presented as an example of a single unified design of the entire character stream processing interface. The sections which follow contain discussion of character set considerations, character stream processing objectives, character stream reduction to canonical form, line and print position deletion, and other interface problems. An appendix gives a formal description of the canonical form for stored character strings used in Multics.

CHARACTER SET CONSIDERATIONS

Although for many years computer specialists have been willing to accept whatever miscellaneous collection of characters and codes their systems were delivered with, and to invent ingenious compromises when designing the syntax of programming languages, the impact of today's computer system is felt far beyond the specialist, and computer printout is (or should be) received by many who have neither time nor patience to decode information printed with inadequate graphic versatility. Report generation has, for some time, been a routine function. Recently, on-line documentation aids, such as RUNOFF,[3] Datatext (IBM Corp.) or RAES (General Electric Co.) have attracted many users. Especially for the latter examples it is essential to have a character set encompassing both upper and lower case letters. Modern programming languages can certainly benefit from availability of a variety of special characters as syntactic delimiters, the ingenuity of PL/I in using a small set notwithstanding.

Probably the minimum character set acceptable today is one like the USASCII 128-character set[4] or IBM's EBCDIC set with the provision that they be fully supported by upper/lower case printer and terminal hardware. The definition of support of a character set is almost as important as the fact of support. To be fully useful, one should be able to use the same full character set in composing program or data files, in literal character strings of a programming language, in arguments of calls to the supervisor and to library routines requiring symbolic names, as embedded character strings in program linkage information, and in input and output to typewriters, displays, printers, and cards. However, it may be necessary to place conversion packages in the path to and from some devices since it is rare to find that all the different hardware devices attached to a system use the same character set and character codes.

* Multics is a comprehensive general purpose time-sharing system implemented on the General Electric 645 computer system. A general description of Multics can be found in Reference 1 or 2.

TABLE I—Escape conventions for input and output of USASCII from an EBCDIC typewriter

ASCII Character Name	ASCII Graphic	Normal EBCDIC Escape	Alternate "edited" Escape
Right Square Bracket	]	¢>	≠
Left Square Bracket	[	¢<	¬≠
Right Brace	}	¢)	⊣
Left Brace	{	¢(	⊢
Tilde	~	¢t	⊢
Grave Accent	`	¢'	

CHARACTER STREAM PROCESSING CONSIDERATIONS

The treatment of character stream input and output may be degraded, from a human engineering point of view, unless it is tempered by the following two considerations:

1. If a computer system supports a variety of terminal devices (Multics, for example, supports both the IBM Model 2741[5] and the Teletype Model 37[6]) then it should be possible to work with any program from any terminal.
2. It should be possible to determine from the printed page, without ambiguity, both what went into the computer program and what the program tried to print out.

To be fully effective, these two considerations must apply to all input and output to the system itself (e.g., when logging in, choosing subsystems, etc.) as well as input and output from user programs, editors, etc.

As an example of the "device independence" convention, Multics uses the USASCII character set in all internal interfaces and provides standard techniques for dealing with devices which are non-USASCII. When using the GE-645 USASCII line printer or the Teletype Model 37, there is no difficulty in accepting any USASCII graphic for input or output from any user or system program. In order to use non-USASCII hardware devices, one USASCII graphic (the left slant) is set aside as a "software excape" character. When a non-USASCII device (say the IBM 2741 typewriter with an EBCDIC print element) is to be used, one first makes a correspondence, so far as possible, between graphics available on the device and graphics of USASCII, being sure that some character of the device corresponds to the software escape character. Thus, for the IBM 2741, there are 85 obviously corresponding graphics; the EBCDIC overbar, cent sign, and apostrophe can correspond to the USASCII circumflex, left slant, and acute accent respectively, leaving the IBM 2741 unable to represent six USASCII graphic characters. For each of the six missing characters a two character sequence beginning with the software escape character is defined, as shown in Table I. The escape character itself, as well as any illegal character code value, is represented by a four character sequence, namely the escape character followed by a 3-digit octal representation of the character code. Thus, it is possible from an IBM 2741 to easily communicate all the characters in the full USASCII set.

A similar, though much more painful, set of escape conventions has been devised for use of the Model 33 and 35 Teletypes. The absence of upper and lower case distinction on these machines is the principal obstacle; two printed 2-character escape sequences are used to indicate that succeeding letters are to be interpreted in a specific case shift.

Note that consideration number two above, that the printed record be unambiguous, militates against character set extension conventions based on non-printing and otherwise unused control characters. Such conventions inevitably lead to difficulty in debugging, since the printed record cannot be used as a guide to the way in which the input was interpreted.

The objective of typewriter device independence also has some implications for control characters. The Multics strategy here is to choose a small subset of the possible control characters, give them precise meanings, and attempt to honor those meanings on every device, by interpretation if necessary. Thus, a "new page" character appears in the subset; on a Model 37 teletype it is interpreted by issuing a form feed and a carriage return; on an IBM 2741 it is interpreted by giving an appropriate number of new line characters.*

Of the 33 possible USASCII control characters, 11 are defined in Multics as shown in Table II.

Red and black shift characters appear in the set because of their convenience in providing emphasis in comments, both by system and by user routines. The half-line forward and half-line reverse feed characters were included to facilitate experimentation with the Model 37 Teletype; these characters are not currently interpretable on other devices.

One interesting point is the choice of a "null" or "padding" character used to fill out strings after the last meaningful character. By convention, padding characters appearing in an output stream are to be discarded, either by hardware or software. The USASCII choice of code value *zero* for the null character has the

*This interpretation of the form feed function is consistent with the International Standards Organization option of interpreting the "line feed" code as "new line" including carriage return.

interesting side effect that if an uninitialized string (or random storage area) is unintentionally added to the output stream, all of the zeros found there will be assumed nulls, and discarded, possibly leaving no effect at all on the output stream. Debugging a program in such a situation can be extraordinarily awkward, since there is no visible evidence that the code manipulating the offending string was ever encountered.

In Multics, this problem was considered serious enough that the USASCII character "delete" (all bits *one*) was chosen as the padding character code. The *zero* code is considered illegal, along with all other unassigned code values, and is printed in octal whenever encountered.

As an example of a control function not appearing in the character set, the printer-on/printer-off function (to allow typing of passwords) is controlled by a special call which must be inserted before the next call to read information. This choice is dictated by the need to get back a status report which indicates that for the currently attached device, the printer cannot be turned on and off. Such a status report can be returned as an error code on a special call; there would be no convenient way to return such status if the function were controlled by a character in the output stream.**

TABLE II—USASCII Control Characters as Used in Multics

USASCII NAME	MULTICS NAME	MULTICS MEANING
BEL	BEL	Sound an audible alarm.
BS	BS	Backspace. Move carriage back one column. The backspace implies overstriking rather than erasure.
HT	HT	Horizontal Tabulate. Move carriage to next horizontal tab stop. Default tab stops are assumed to be at columns 11, 21, 31, 41, etc.
LF	NL	New Line. Move carriage to left edge of next line.
SO	RRS	Red Ribbon Shift.
SI	BRS	Black Ribbon Shift.
VT	VT	Vertical Tabulate. Move carriage to next vertical tab stop. Default tab stops are assumed to be at lines 11, 21, 31, etc.
FF	NP	New Page. Move carriage to the left edge of the top of the next page.
DC2	HLF	Half-Line Forward Feed.
DC4	HLR	Half-Line Reverse Feed.
DEL	PAD	Padding Character. This character is discarded when encountered in an output line.

CANONICAL FORM FOR STORED CHARACTER STRINGS

Probably the most significant impact of the constraint that the printed record be unambiguous is the interaction of that constraint with the carriage motion control characters of the USASCII and EBCDIC sets. Although most characters imply "type a character in the current position and move to the next one," three commonly provided characters, namely backspace, horizontal tab, and carriage return (no line feed) do cause ambiguity.

For example, suppose that one chooses to implement an ALGOL language in which keywords are underlined. The keyword for may now be typed in at least a dozen different ways, all with the same printed result but all with different orders for the individual letters and backspaces. It is unreasonable to expect a translator to accept a dozen different, but equivalent, ways of typing every control word; it is equally unreasonable to require that the typist do his underlining in a standard way since if he slips, there is no way he can tell from his printed record (or later protestations of the compiler) what he has done wrong. A similar dilemma occurs in a manuscript editing system if the user types in underlined words, and later tries to edit them.

An answer to this dilemma is to process all character text entering the system to convert it into a *canonical form.* For example, on a "read" call Multics would return the string:

$$\_\langle BS\rangle f\_\langle BS\rangle o\_\langle BS\rangle r$$

(where $\langle BS\rangle$ is the backspace character) as the canonical character string representation of the printed image of for independently of the way in which it had been typed. Canonical reduction is accomplished by scanning across a completed input line, associating a carriage position with each printed graphic encountered, then sorting the graphics into order by carriage or print position. When two or more graphics are found in the same print position, they are placed in order by numerical collating sequence with backspace characters between. Thus, if two different streams of characters produce the same printed image, after canonical reduction they will be represented by the same stored string. Any program can thus easily compare two canonical strings to discover if they produce the same printed image. No restriction is

** The initial Multics implementation temporarily uses the character codes for USASCII ACK and NAK for this purpose, as an implementation expedient. In addition, a number of additional codes are accepted to permit experimentation with special features of the Model 37 Teletype; these codes may become standard if the features they trigger appear useful enough to simulate on all devices.

placed on the human being at his console; he is free to type a non-canonical character stream. This stream will automatically be converted to the canonical form before it reaches his program. (There is also an escape hatch for the user who wants his program to receive the raw input from his typewriter, unprocessed in any way.)

Similarly, a typewriter control module is free to rework a canonical stream for output into a different form if, for example, the different form happens to print more rapidly or reliably.

In order to accomplish canonical reduction, it is necessary that the typewriter control module be able to determine unambiguously what precise physical motion of the device corresponds to the character stream coming from or going to it. In particular, it must know the location of physical tab settings. This requirement places a constraint on devices with movable tab stops; when the tab stops are moved, the system must be informed of the new settings.

The apparent complexity of the Multics canonical form, which is formally described in Appendix I, is a result of its generality in dealing with all possible combinations of typewriter carriage motions. Viewed in the perspective of present day language input to computer systems, one may observe that many of the alternatives are rarely, if ever, encountered. In fact for most input, the following three statements, describing a simplified canonical form, are completely adequate:

1. A message consists of strings of character positions separated by carriage motion.
2. Carriage motions consist of New Line or Space characters.
3. Character positions consist of a single graphic or an overstruck graphic. A character position representing overstrikes contains a graphic, a backspace character, a graphic, etc., with the graphics in ascending collating sequence.

Thus we may conclude that for the most part, the canonical stream will differ little with the raw input stream from which it was derived.

A strict application of the canonical form as given in Appendix I has a side effect which has affected its use in Multics. Correct application leads to replacement of all horizontal tab characters with space characters in appropriate numbers. If one is creating a file of tabular information, it is possible that the ambiguity introduced by the horizontal tab character is in fact desirable; if a short entry at the left of a line is later expanded, words in that entry move over, but items in columns to the right of that entry should stay in their original carriage position; the horizontal tab facilitates expressing this concept. A similar comment applies to the form feed character.

The initial Multics implementation allows the horizontal tab character, if typed, to sneak through the canonical reduction process and appear in a stored string. A more elegant approach to this problem is to devise a set of conventions for a text editor which allows one to type in and edit tabular columns conveniently, even though the information is stored in strictly canonical form. Since the most common way of storing a symbolic program is in tabular columns, the need for simple conventions to handle this situation cannot be ignored.

It is interesting to note that most format statement interpreters, such as those commonly implemented for FORTRAN and PL/I, fail to maintain proper column alignment when handed character strings containing embedded backspaces, such as names containing overstruck accents. For complete integration of such character strings into a system, one should expand the notion of character counts to include print position counts as well as storage position counts. For example, the value returned by a built-in string length function should be a print position count if the result is used in formatting output; it should be a storage location count if the result is used to allocate space in memory.

LINE AND PRINT POSITION DELETION CONVENTIONS

Experience has shown that even with sophisticated editor programs available, two minimal editing conventions are very useful for human input to a computer system. These two conventions give the typist these editing capabilities at the instant he is typing:

1. Ability to delete the last character or characters typed.
2. Ability to delete all of the current line typed up to the point.

(More complex editing capabilities must also be available, but they fall in the domain of editing programs which can work with lines previously typed as well as the current input stream.) By framing these two editing conventions in the language of the canonical form, it is possible to preserve the ability to interpret unambiguously a typed line image despite the fact that editing was required.

The first editing convention is to reserve one graphic, (in Multics, the "number" sign), as the *erase* character. When this character appears in a print position, it erases itself and the contents of the previous print position. If the erase follows simple carriage motion, the entire carriage motion is erased. Several successive

erase characters will erase an equal number of preceding print positions or simple carriage motions. Since erase processing occurs after the transformation to canonical form, there is no ambiguity as to which print position is erased; the printed line image is always the guide. Whenever a print position is erased, the carriage motions (if any) on the two sides of the erased print position are combined into a single carriage motion.

The second editing convention reserves another graphic (in Multics, the "commercial at" sign) as the *kill* character. When this character appears in a print position, the contents of that line up to and including the kill character are discarded. Again, since the kill processing occurs after the conversion to canonical form, there can be no ambiguity about which characters have been discarded. By convention, kill is done before erase, so that it is not possible to erase a kill character.

OTHER INTERFACE CONVENTIONS

Two other conventions which can smooth the human interface on character stream input and output are worth noting. The first is that many devices contain special control features such as line feed without carriage movement, which can be used to speed up printing in special cases. If the system-supplied terminal control software automatically does whatever speedups it can identify, the user is not motivated to try to do them himself and risk dependence on the particular control feature of the terminal he happens to be working with. For example, the system can automatically insert tabs (followed by backspaces if necessary) in place of long strings of spaces, and it also can type centered short tabular information with line feed and backspace sequences between lines.

The second convention has been alluded to already. If character string input is highly processed for routine use, there must be available an escape by which a program can obtain the raw, unconverted, unreduced and unedited keystrokes of the typist, if it wants to. Only through such an escape can certain special situations (including experimenting with a different set of proposed processing conventions) be handled. In Multics, there are three modes of character handling—normal, raw, and edited.* The raw mode means no processing whatsoever on input or output streams, while the normal mode provides character escapes, canonical reduction, and erase and kill editing. The edited mode (effective only on output if requested) is designed to produce high quality, clean copy; every effort is made to avoid using escape conventions. For example, illegal characters are discarded and graphics not available on the output device used are typed with the "overstrike" escapes of Table I, or else left as a blank space so that they may be drawn in by hand.

CONCLUSIONS

The preceding sections have discussed both the background considerations and the design of the Multics remote terminal character stream interface. Several years of experience in using this interface, first in a special editor on the 7094 Compatible Time-Sharing System and more recently as the standard system interface for Multics, have indicated that the design is implementable, usable and effective. Probably the most important aspect of the design is that the casual user, who has not yet encountered a problem for which canonical reduction, or character set escapes, or special character definitions are needed, does not need to concern himself with these ideas; yet as he expands his programming objectives to the point where he encounters one of these needs, he finds that a method has been latently available all along in the standard system interface.

There should be no assumption that the particular set of conventions described here is the only useful set. At the very least, there are issues of taste and opinion which have influenced the design. More importantly, systems with only slightly different objectives may be able to utilize substantially different approaches to handling character streams.

ACKNOWLEDGMENTS

Many of the techniques described here were developed over a several year time span by the builders of the 7094 Compatible Time-Sharing System (CTSS) at MIT Project MAC, and by the implementers of Multics, a cooperative research project of the General Electric Company, the Bell Telephone Laboratories, Inc., and the Massachusetts Institute of Technology.

The usefulness of a canonical form for stored character strings was independently brought to our attention by E. Van Horne and C. Strachey; they had each implemented simple canonical forms on CTSS and in the TITAN operating system for the ATLAS computer, respectively. F. J. Corbató and R. Morris developed the pattern of escape sequence usage described here. Others contributing to the understanding of the issues involved in the character stream interface were R. C. Daley, S. D. Dunten, and M. D. McIlroy.

Work reported here was supported in part by the advanced Research Projects Agency, Department of

*The "raw" mode is not yet implemented.

Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction is permitted for any purpose of the United States Government.

REFERENCES

1 F J CORBATÓ et al
A new remote-accessed man-machine system
AFIPS Conference Proceedings 27 1965 FJCC Spartan Books Washington D C 1965 pp 185–247

2 *The multiplexed information and computing service: Programmer's manual*
M I T Project MAC Cambridge Massachussetts 1969 To be published

3 J H SALTZER
Manuscript typing and editing
In The Compatible Time-Sharing Sytem: A Programmer's Guide 2nd Edition M I T Press Cambridge Massachusetts 1965

4 *USA standard code for information interchange*
X3 4-1968 USA Standards Institute October 1968

5 *IBM 2741 communications terminal*
IBM Systems Reference Library Form A24-3415 IBM Corporation New York

6 *Model 37 teletypewriter stations for DATA-PHONE service*
Bell System Data Communications Technical Reference American Telephone and Telegraph Company New York September 1968

7 *PL/I language specifications*
IBM System Reference Library Form C28-6571 IBM Corporation New York

APPENDIX I

The Multics canonical form

To describe the Multics canonical form, we give a set of definitions of a canonical message. Each definition is followed by a discussion of its implications. PL/I-style formal definitions are included for the benefit of readers who find them useful.[7] Other readers may safely ignore them at a small cost in precision. In the formal definitions, capitalized abbreviations stand for the control characters in Table II.

1. The canonical form deals with messages. A message consists of a sequence of print positions, possibly separated by, beginning, or ending with carriage motion.

message : : = [carriage motion]
[[print position]···[carriage motion]]···

Typewriter input is usually delimited by *action* characters, that is, some character which, upon receipt by the system, indicates that the typist is satisfied with the previous string of typing. Most commonly, the new line character, or some variant, is used for this function. Receipt of the action character initiates canonical reduction.

The most important property on the canonical form is that graphics are in the order that they appear on the printed page reading from left to right and top to bottom. Between the graphic characters appear only the carriage motion characters which are necessary to move the carriage from one graphic to the next. Overstruck graphics are stored in a standard form including a backspace character (see below).

2. There are two mutually exclusive types of carriage motion, gross motion and simple motion.

$$\text{carriage motion : : =} \left\{\begin{matrix}\text{gross motion} \\ \text{simple motion} \\ \text{gross motion simple motion}\end{matrix}\right\}$$

Carriage motion generally appears between two graphics; the amount of motion represented depends only on the relative position of the two graphics on the page. Simple motion separates characters within a printed line; it includes positioning, for example, for superscripts and subscripts. Gross motion separates lines.

3. Gross motion consists of any number of successive New Line (NL) characters.

$$\text{gross motion : : =} \{\text{NL}\}\cdots$$

The system must translate vertical tabs and form feeds into new line characters on input.

4. Simple motion consists of any number of Space characters (SP) followed by some number (possibly zero) of vertical half-line forward (HLF) or reverse (HLR) characters. The number of vertical half line feed characters is exactly the number needed to move the carriage from the lowest character of the preceding print position to the highest character of the next print position.

$$\text{simple motion : : =} \{\text{SP}\}\cdots\left[\begin{matrix}[\text{HLF}]\cdots \\ \\ [\text{HLR}]\cdots\end{matrix}\right]$$

The basis for the amount of simple carriage motion represented is always the horizontal and vertical distance between successive graphics that appears on the actual device. In the translation to and from the canonical form, the system must of course take into account the actual (possibly variable) horizontal tab stops on the physical device.

In some systems, a "relative horizontal tab" character is defined. Some character code (for example, USASCII DC1) is reserved for this meaning, and by convention the immediately following character storage position contains a count which is interpreted as the size of the horizontal white space to be left. Such a character fits smoothly into the canonical form de-

scribed here in place of the successive spaces implied by the definition above. It also minimizes the space requirement of a canonical string. It does require some language features, or subroutines, to extract the count as an integer, to determine its size. It also means that character comparison is harder to implement; equality of a character with one found in a string may mean either that the hoped for character has been found or it may mean that a relative tab count happens to have the same bit pattern as the desired character; reference to the previous character in the string is required to distinguish the two cases.

5. A print position consists of some non-zero number of character positions, occupying different half line vertical positions in the same horizontal carriage position. All but the last character position of a print position are followed by a backspace character and some number of HLF characters.

print position : : = character position
[BS [HLF]···character position]···

6. A character position consists of a sequence of graphic formers separated by backspace characters. The graphic formers are ordered according to the USASCII coded numeric value of the graphics they contain. (The first graphic former contains the graphic with the smallest code, etc.) Two graphic formers containing the same graphic will never appear in the same character position.

character position : : = graphic former
[BS graphic former]···

Note that all possible uses of a backspace character in a raw input stream have been covered by statements about horizontal carriage movements and overstruck graphics.

7. A graphic former is a possibly zero-length setup sequence of graphic controls followed by one of the 94 USASCII non-blank graphic characters.

graphic former : : = [setup sequence] { one of the 94 UASCII graphic characters }

8. A graphic setup sequence is a color shift or a bell (BEL) or a color shift followed by a bell. The color shift only appears when the following graphic is to be a different color from the preceding one in the message.

setup sequence : : = { [RRS [BEL] BRS] ; BEL }

in the absence of a color shift, the first graphic in a message is printed in black shift. Other control characters are treated similarly to bell. They appear immediately before the next graphic typed, in the order typed.

By virtue of the above definitions, the control characters HT, VT, and CR will never appear in a canonical stream.

A study of heuristic learning methods for optimization tasks requiring a sequence of decisions*

by L. ROWELL HUESMANN

Yale University
New Haven, Connecticut

INTRODUCTION

Learning is a broad term covering many different phenomena. It is convenient to segment learning into three different problems in induction: the collection and use of stochastic information on past performance in order to improve performance, the determination of which variables are relevant to the decisions being made, and the derivation of performance rules in the predicate calculus from the collected data. This study concentrates on the first problem.

THE ISSUES FOR INVESTIGATION

(a) Can a digital computer program significantly improve its performance on an optimization task of real-world complexity (and generalize that improvement to other problems of the same type) solely through ordinal feedback from intercomparisons of the solutions it has produced?

Most of the previous work in machine learning dealt with pattern recognition or game playing tasks. Yet these tasks have specific characteristics that differentiate their requirements for a learning mechanism from other tasks' requirements. Both are essentially win-loss or right-wrong tasks. In addition, in pattern recognition, feedback about the success of a decision is usually immediate. Yet many tasks have other than binary outcomes—that is, they are optimization tasks or problems in finding the "best" solution, according to some objective criterion, from a set of feasible solutions. Usually, the problem solver does not even know how well he can do. Consumer decisions, social decisions, and business decisions are often problems of this type.

With many optimization tasks one can obtain interval information about the relative worth of two solutions, however for others only an ordinal scale of solutions can be found. More important, it is often an order of mangnitude easier for a program to decide whether one solution is better than another than for it to decide how much better. Hence, it is desirable to find a mechanism that can improve a program's performance solely from ordinal feedback.

(b) Can significant improvement occur if the task environment is characterized for the program by a vector of relevant stimulus variables (a state vector)?

Another characteristic of much of the previous work in machine learning is that most learning mechanisms have combined the stimulus variables in linear polynomials and selected a response on the basis of the various polynomials' values. Many of these schemes are called stimulus voting procedures because each stimulus votes separately for a response.

The limitations of such linear machines are well known and have been analyzed in detail.[1,2] What is particularly disappointing is the simplicity of some patterns that cannot be handled by linear machines. For example, consider the association pattern in Table 1. When the values of the two features are the same, response R1 is required; otherwise, R2 is required. Let us now show that linear discriminant functions cannot be used to make this classification.

Theorem: Linear discriminant functions do not exist for some very simple classifications of features. In particular none exist for the classification shown in Table 1. Proof: The theorem will be proved by assuming the linear discriminant functions do exist and finding a

* This paper is based on a Ph.D. thesis completed at Carnegie-Mellon University, Pittsburgh, Pa. The project was supported in part by United States Public Health Service grants MH-07722 and MH-30,606-01A1. The author is indebted to Mr. Herbert A. Simon for his advice and assistance.

TABLE I—A Simple Discrimination That Is Not Realizable with Linear Discriminant Functions

VALUES OF FEATURES		*DESIRED RESPONSES*	
(F_1) *FEATURE 1*	(F_2) *FEATURE 2*	(R_1) *RESPONSE 1*	(R_2) *RESPONSE 2*
	1	X	
1			
	2		X
	1		X
2			
	2	X	

contradiction. Let E_1 be the linear discriminant function for R_1

$$E_1 = C_{11}F_1 + C_{12}F_2$$

Let E_2 be the linear discriminant function for R_2

$$E_2 = C_{21}F_1 + C_{22}F_2$$

If linear discriminant functions exist that can make this discrimination, then

For ($F_1 = 1, F_2 = 2$) and ($F_1 = 2, F_2 = 1$)

$$E_2 - E_1 = C_{21}F_1 + C_{22}F_2 - C_{11}F_1 - C_{12}F_2 > 0 \quad (1.1)$$

For ($F_1 = 1, F_2 = 1$) and ($F_1 = 2, F_2 = 2$)

$$E_1 - E_2 = C_{11}F_1 + C_{12}F_2 - C_{21}F_1 - C_{22}F_2 > 0 \quad (1.2)$$

Substituting the values of the features gives from (1.1)

$$C_{21} + 2C_{22} - C_{11} - 2C_{12} > 0 \quad (1.3)$$

and

$$2C_{21} + C_{22} - 2C_{11} - C_{12} > 0 \quad (1.4)$$

and from (1.2)

$$C_{11} + C_{12} - C_{21} - C_{22} > 0 \quad (1.5)$$

$$-(C_{11} + C_{12} - C_{21} - C_{22}) < 0$$

$$C_{22} + C_{21} - C_{11} - C_{12} < 0 \quad (1.6)$$

But adding (1.3) and (1.4) gives

$$3C_{22} + 3C_{21} - 3C_{12} - 3C_{11} > 0$$

$$C_{22} + C_{21} - C_{11} - C_{12} > 0 \quad (1.7)$$

Equation (1.7) contradicts (1.6). Since the conclusion of a correct line of reasoning has been a contradiction, the assumption that a linear discriminant function exists must be false, and the theorem is proved.

A mechanism that associated *states* of the environment with strategies or responses could learn such discriminations. Unfortunately, a state vector description requires a great deal more computer storage. For example, for R stimulus variables and N values per variable a parsimonious representation of the stimulus state requires on the order of N^R storage cells. On the other hand, one needs only $N*R$ cells to represent the status of each stimulus variable independently of the other variables and only R cells to represent the stimulus situation as the value of a linear polynomial. However, psychological evidence indicates that humans seldom attend to more than a few environmental features at a time[3] so a state-vector of low dimensionality might be a reasonable representation for a learning program. This is the representation we adopted.

The learning problem

We view the learning problem as one of associating states of the environment, defined by some set of stimulus variables to which the problem solver is attending, with strategies for performance. The strength of such associations can be represented by the entries in a table of connections or matrix whose rows represent stimulus states and whose columns represent strategies. We want to see if significant learning can be accomplished on a very complex optimization task if the stimulus environment is represented by a state vector of some of the most obvious relevant stimulus variables and only ordinal feedback is used.

THE TASK TO BE LEARNED

To avoid spending a majority of the programming effort on a performance program for solving a very general class of optimization tasks, it was decided to restrict the study to one specific task, the project scheduling task. A sample problem for this task is shown in Figure 1. The objective is to complete all the jobs in as short a time as possible by executing them in parallel. It is a difficult real-world task faced by management scientists, but it can be shown to be very similar to other optimization tasks requiring a sequential set of decisions, e.g., finding the minimum number of moves to checkmate, or the Traveling Salesman Problem. A task very similar to the project scheduling task was used by Fisher and Thompson[4] in a study that suggested the learning technique we have used. One can view optimization tasks that require a sequence of decisions as problems in finding the shortest or longest path through a decision tree. A feasible solution is any path from the root of the tree to a terminal or goal node. The branches descending from a node represent possible decisions and the nodes represent the status of the "system" after a decision is made.

THE LEARNING TECHNIQUE

Given a state-vector representation of the task environment and a set of performance strategies, the learning mechanism must create a good (and generalizable) table of connections between stimulus states and strategies. An informal "hill climbing" procedure will be used to construct the table. Viewing learning as constructing a table of connections is not a new idea.[5] However, unlike almost all previous learning programs, this one will have no way to make an absolute judgment about the utility of a solution. Since the problems to be attacked are optimization problems themselves, the learning program cannot determine when it has achieved the best solution. How will feedback be obtained?

The best previous solution will be designated as a bench mark solution and new solutions will be compared to it. If the new solution is better, the comparison is positive; if it is worse, the comparison is negative. A fairly sophisticated comparison procedure was developed to make comparisons feasible as frequently as possible during construction of a solution. Hence, one comparison corresponds to what is normally called one trial in the learning literature and one trial on a problem includes a whole series of comparisons. One can show that this technique can be applied (with a few restrictions) to almost any optimization task requiring sequential decisions.

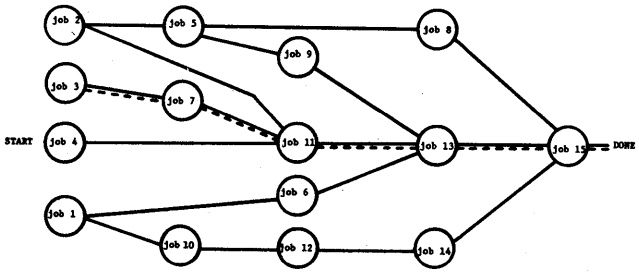


Figure 1—A sample project scheduling problem. The leftmost jobs can be executed initially. The lines indicate the prerequisities for the other jobs. No job can be executed until all the jobs connected to it from the left are completed. The dotted line is the critical path for the execution times given below. This problem has only one resource need limiting how many jobs can be scheduled in parallel.

Job	*Units of Time Needed*	*Units of Resource Occupied*
job 1	4	5
job 2	3	5
job 3	2	1
job 4	3	3
job 5	4	6
job 6	5	4
job 7	3	4
job 8	3	2
job 9	3	4
job 10	3	6
job 11	6	6
job 12	8	4
job 13	7	4
job 14	1	6
job 15	1	10

THE DESIGN OF THE PROGRAM

This program, like most learning programs, should be viewed as two closely interacting routines—a performance routine and a learning routine. The routines were written in IPL-5. Let us first discuss the performance routine.

The performance routine

This routine is designed to find the shortest path through the tree of feasible solutions, i.e., feasible job schedules. Each level in the tree corresponds to a different time; each node in the tree specifies what jobs are completed, what jobs are currently being executed, and what jobs remain to be scheduled; each branch indicates the scheduling of a particular set of jobs. Hence, two geometrically different nodes may have the same meaning with different histories. Every path eventually leads to a node specifying that all jobs are complete. The objective of the performance program is to find a path through this tree that ends at the highest level terminal node (minimum time path).

The performance program uses a "depth first" approach to search. It looks ahead along a path through the tree until it detects a node where the path can be evaluated. Of course, there will be no evaluations during the production of the initial solution since there is no solution for comparison. At each node encountered in the look ahead process, the program must decide what branch to follow next. This is equivalent to choosing the jobs that should be scheduled at that time. When a node is reached that can be evaluated, the learning program is called in to compare the current path with the bench mark solution. The comparison is either indeterminate, positive—the new path is more desirable, or negative—the present solution is more desirable. If the comparison is indeterminate or positive, the performance routine looks ahead deeper along the current path. In addition, when the evaluation is positive, the current path is merged with the bench mark solution to form a new bench mark solution. On the

other hand, if the evaluation is negative, the performance routine abandons search along the current path. It may either return to the top of the solution tree and investigate a new path or look ahead deeper from the corresponding but preferred node on the present bench mark solution's path. By "corresponding node" we mean the node on the solution path that was used in the evaluation.

The tables of connections

In carrying out this procedure the performance program has to make two types of decisions. As mentioned above, following a negative comparison, the program must decide whether to back up to the start or continue from a point on the present bench mark solution; the program must also decide what branch to follow (what jobs to schedule) at each node encountered in the look ahead process. While the former of these decisions requires a general-problem-solving strategy, the latter decision requires a task-specific strategy. To select these strategies, the performance routine employs two tables of connections. One table links a state vector composed of characteristics of the current search situation to general strategies, in this case strategies specifying what to do after a negative comparison. The other table connects a state vector of task relevant variables to a set of task specific strategies, in this case job selection strategies.

Both of these tables are represented by tree structures in the computer's memory. A numerical value associated with Strategy i at State-node j will provide a measure of the past success of that strategy in State j relative to the success of other strategies in that state.

Selecting a strategy

The information in a state node could be used in any of several ways to select a strategy. For example, if one wishes to select a good strategy, one might choose the strategy whose success value is the greatest of all the values at the node, or one might select a strategy probabilistically in proportion to the success values. On the basis of several pilot runs it was decided that the performance program should construct the initial solution on each run by selecting the strategies with the highest success values (ties are broken randomly). During the rest of the run the performance routine would select strategies probabilistically. Specifically, the probability of choosing Strategy s_i whose success value in the current state is v_i, from n strategies whose values in the current state are $v_1 \ldots v_n$ is given by

$$P(s_i) = v_i \Big/ \sum_{k=1}^{n} v_k$$

Built-in heuristics

The performance routine was not intended to begin as a completely naive problem solver. Certain general and task specific heuristics were built into the routine while other heuristics were introduced via the initial entries in the general-strategy table of connections. These heuristics were ones that most human problem solvers would have learned before ever attempting a problem of this type or ones that would be suggested by a cursory glance at the literature on the task. Foremost, among the general heuristics built into the program, is sub-goal evaluation. During the look-ahead process, the performance program asks the learning program to evaluate virtually every potential sub-goal—that is, every node on the current path that is on the same level as a node of the solution path. (In look-ahead searches on a typical problem twenty-eight sub-goals were evaluated for every evaluation of a complete path through the tree of feasible solutions.) A second built-in general heuristic is the program's "next event" approach to search. During the look-ahead process many nodes are encountered where no decisions need to be made. For example, no jobs can be scheduled at a node unless a job terminates there. Hence, the performance program jumps from node to node ignoring intervening nodes where no decisions need to be made. This heuristic speeds performance greatly, but it is dependent upon another heuristic—a task specific heuristic—included in the performance program. The performance routine always schedules as many jobs as resource constraints permit; so no new job can be scheduled until a job terminates and frees some resources. Such a heuristic is not without its drawbacks. There are a few situations where it prevents the program from searching a slightly superior branch. However, it is a heuristic with strong intuitive appeal, one that reduces the number of branches in the solution tree considerably, and one that permits implementation of the next event search process reducing the number of nodes to be analyzed during look-ahead.

Three heuristics dealing with search behavior were introduced through the initial values of the success terms in the general-strategy table of connections. These heuristics deal with what the program should do following the discovery that a branch presently being searched can only be inferior to the solution (negative comparison). The probability of "backing

up" to the start was made an inverse function of the depth that search had progressed into the solution tree and a direct function of the number of dead end branches encountered (negative comparisons and non-positive comparisons of complete paths). Thirdly, whichever "back up" strategy is selected, it should be tried several times consecutively before being abandoned.

Storing a solution

The performance program remembers only the present solution path and the path currently being searched. Both are stored as lists of scheduled jobs separated by time markers. The jobs preceding the nth time marker are the jobs being executed at time n. Associated with the list representing the path currently being searched is a list of the values in the tables of connections that have been selected during the search—that is, the values corresponding to the state-strategy pairings used to produce the path. Whenever a strategy is used, its value in the current state is added to this list. Hence, all the cells that the learning routine will modify are contained on this list.

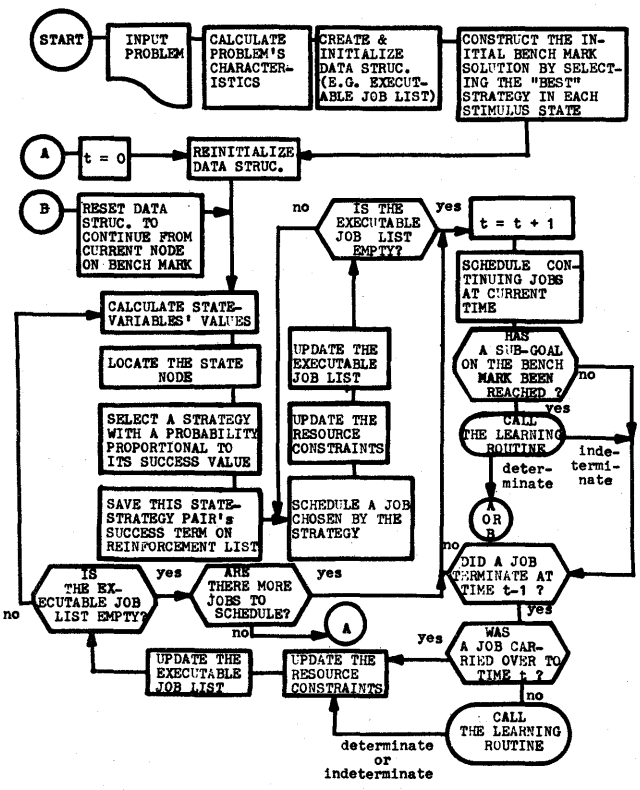


Figure 2—A flow chart of the performance routine.

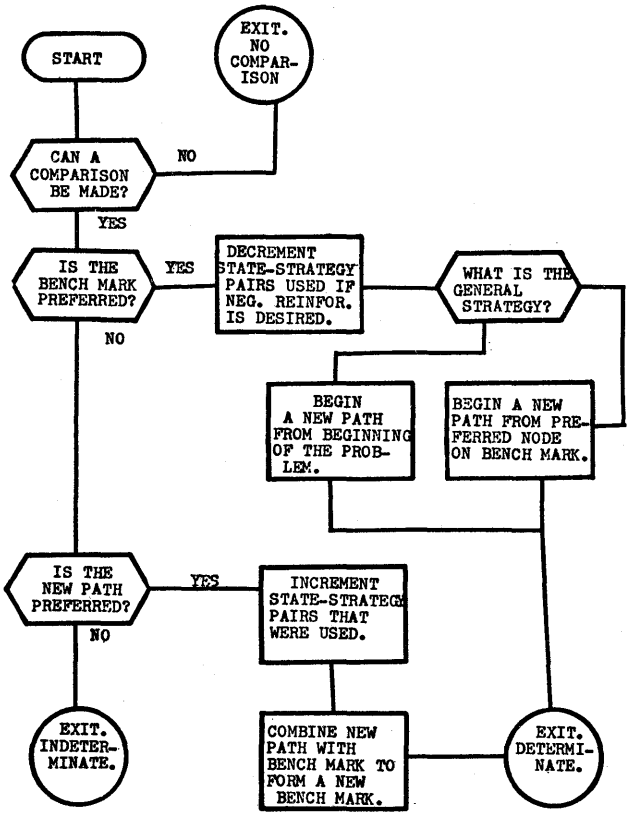


Figure 3—A flow chart showing the steps the learning routine takes when called upon to compare part of a new solution with the bench mark solution.

A gross flow chart of the performance procedure is presented in Figure 2.

The learning routine

The learning program evaluates paths through the tree of feasible solutions by comparing them with the bench mark solution (best solution so far), and it alters the tables of connections on the basis of these evaluations.

Comparing solutions

How does the program determine which of two paths is preferred? Path Z1 up to node x is preferred to path Z2 up to node y if node x and y are at the same level in the tree of feasible solutions and if node x "dominates" node y. A node, one should remember, specifies the set of jobs currently completed and the set of jobs currently being executed. To say node x on Z1 dominates node y on Z2 means that (a) all jobs

TABLE II—These Sample Schedules Illustrate the Construction of a New and Superior Bench Mark Solution (*Z4*) Out of the Old Bench Mark (*Z2*) and a New Partial Solution (*Z1*).

	SCHEDULES			
TIME	*Z1*	*Z2*	*Z3*	*Z4*
1	job 1 job 2	job 1 job 4	job 1 job 2	job 1 job 2
2	job 3 job 4	job 2 |	job 3 job 4	job 3 job 4
3	| |	job 3 |	| |	| |
4	job 5 |	|	job 5 |	job 5 |
5		job 5	job 5	job 6 job 7
6		job 6 job 7	job 6 job 7	job 8 |
7		job 8 |	job 8 |	|
8		|	|	

LIST REPRESENTATION OF Z1:

Z1—0
job 1
job 2
time mark
job 3
job 4
time mark
job 3
job 4
time mark
job 4
job 5
time mark—0

scheduled on Z1 prior to x are completed by x, (b) all jobs on Z2 that are completed by y or being executed at y are completed by x on Z1 (from (a), any job *on* Z1 prior to x is completed by x), and (c) there is at least one job completed on Z1 prior to x that is not on Z2 prior to y. From this definition if any node x on Z1 dominates its corresponding node y on Z2 (the node at the same level), then combining Z1 prior to x with Z2 after y produces a new path at least as short as Z2 and in most cases shorter. Hence, Z1 prior to x is preferred to Z2 prior to y. One should be careful to clearly understand these statements as they are essential to the learning method. They form a task specific algorithm for judging partial schedules. To verify that the new path will indeed be no longer, one simply recognizes that Z2 after y can always be added onto Z1 before x without any changes since all jobs on Z1 are completed at x. Furthermore, at least one job on Z2 after y has already been executed and can be deleted. If this deletion (or deletions) shortens Z2 after y the new path will be shorter. Consider as an example the schedules Z1 and Z2 in Table 2. At time 4 the node on Z1 dominates the corresponding node on Z2. As a result Z1 and Z2 can be combined into the new schedule Z3. By deleting "job 5" which had already been completed on Z1 and moving the other jobs up in time, a shorter schedule Z4 was then produced.

The large majority of evaluations turn out to be indeterminate. For example, during training on a typical problem about 94% were indeterminate. When the comparison is negative (the present bench mark solution is preferred), the tables of connections are not altered and control is returned to the performance routine which decides whether to look ahead from the corresponding node of the bench mark or back up to the top of the tree. When the comparison is positive (the current path is preferred), the learning program alters the tables of connections and constructs a new bench mark solution.

Altering the memory structures

Altering the tables of connections is fairly trivial. Remember that during look ahead a list is maintained of all the success terms associated with the selected state strategy pairings. This requires very little storage, only one cell for each decision made since the last

positive or negative comparison. To positively reinforce the state-strategy pairings participating in the construction of a better solution, each element of this list is simply incremented. On the basis of pilot studies we selected an increment of 3 over smaller values. Larger values might produce more rapid learning but also less stable. Obviously, an entire study could be devoted to finding the optimal value for this increment. With an increment of 3 the probability of selecting each strategy is altered as follows.

Let $P_t(s_j/R)$ be the probability at time t of selecting strategy s_j in state R.
v_j be the success value associated with the jth strategy in state R at time t.
n be the total number of strategies. Then, as mentioned earlier,

$$P_t(s_i/R) = v_i \Big/ \left(\sum_{j=1}^{n} v_j\right)$$

and if strategy s_i is reinforced in state R at time t

$$P_{t+1}(s_i/R) = (v_i + 3) \Big/ \left(\sum_{j=1}^{n} v_j + 3\right)$$

$$P_{t+1}(s_k/R) = v_k \Big/ \left(\sum_{j=1}^{n} v_j + 3\right), \quad k \neq i$$

or letting

$$M = \sum_{j=1}^{n} v_j$$

at time t

$$P_{t+1}(s_i/R) = \frac{\mathrm{M}}{M+3} P_t(s_i/R) + \frac{3}{M+3}$$

$$P_{t+1}(s_k/R) = \frac{\mathrm{M}}{M+3} P_t(s_k/R)$$

These changes will be called positive reinforcement.

Consistent decision making during learning

This completes the description of the learning routine. One very important addition was made to the learning scheme as a result of some early failures in the pilot studies.

Principle: While exploring a path through the tree of feasible solutions, a performance program used with a learning routine should employ the same strategy every time the same state occurs (make the same decision in the same situation) until the path has been successfully evaluated (positively or negatively).

When this principle is not adhered to, credit assignment becomes almost impossible. Conceivably, all the strategies could be used in the same state before an evaluation occurred. In this case the bad strategies may mask the good strategies, and one has no way to distinguish between them. Hence, it is not sufficient to "select a strategy in proportion to its past successes." One must first check to see if a strategy has already been paired with the current state, and, if so, use that strategy.

SELECTING STRATEGIES AND FEATURES

One might well argue that the major portion of this program's work is done by the programmer when he selects the stimulus features for attention and the potential strategies for use. Yet this is exactly what happens to the human beginner. He generally derives his first ideas about strategies and features from a teacher, a book, or his experience with other similar tasks. The features and strategies that we selected for use were simple ones that would occur to anyone who made a cursory glance at the literature on scheduling problems. Within the program the features and strategies were represented as lists of components in such a way that new strategies or features could be synthesized. Later we will see how this learning mechanism could employ its feedback to eliminate poor strategies or features and introduce new ones.

Five task-specific strategies and three features of 3, 3, and 4 values were used initially. Hence, there were 3*4*3 or 36 state nodes in the task-specific table of connections. Each state, of course, really represented a broad class of stimulus situations. With five strategies per node the total storage requirement of the task specific table of connections was only 463 IPL–5 cells or 926 32-bit words. All the success values in this table were initially set at 10. Other smaller values were tried during the pilot studies and found to change the performance routine's behavior too radically in early training.

The general-strategy table of connections used in these experiments was employed only to choose between two search strategies. The "previous-strategy" feature thus had two values while the other two features had three value classes. Hence, there were 18 state nodes requiring 133 IPL–5 cells or 266 32-bit words. This means that the two tables' total storage requirement was 1,192 computer words. The initial success

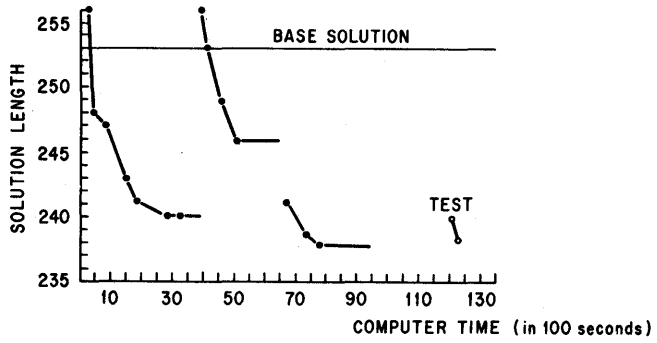


Figure 4.1—The solutions produced during training on Problem 1 with positive reinforcement. The discontinuities are points where all previous solutions were erased from the program's memory, and it began again on the problem from scratch.

values in the general-strategy table were assigned to implement certain heuristics as discussed in the section on the performance routine.

EXPERIMENT I

To answer questions (a) and (b) we tested this program's learning ability on three project scheduling problems.

The dependent variable on which a learning mechanism should be evaluated is improvement in performance not the quality of performance. We want to demonstrate that the proposed learning mechanism, using only ordinal feedback, can learn what strategy to apply in what state so that the performance program performs significantly better on the training problem and on other problems of the same type.

Fifteen project scheduling problems (unbiased in any obvious manner) were generated randomly by the computer to find three that satisfied hardware and complexity constraints (most were too simple).

We will call these Problems 1, 2, and 3. The program was trained on Problem 1, trained more on Problem 2, and tested for ten minutes on Problem 3. Then we retrained the program from scratch on Problem 3 and tested it on Problem 1. No negative reinforcement was applied in this experiment. If the bench mark solution was definitely superior, the new path was abandoned and the program selected a general strategy telling it what to do next.

Results

The training significantly improved the performance program after only a moderate number of positive

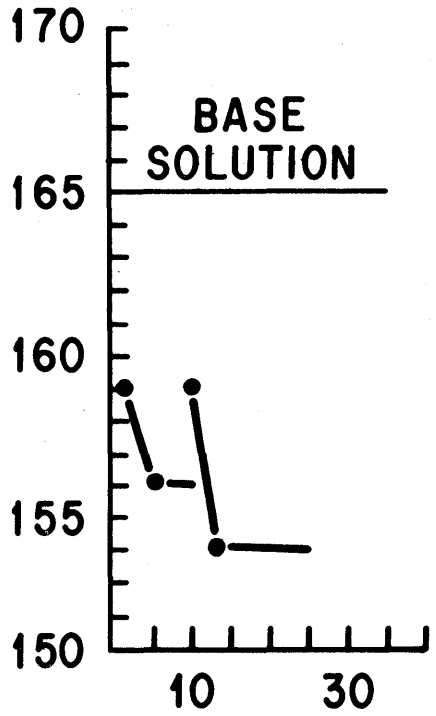


Figure 4.2—The solutions produced during additional training on Problem 2 with positive reinforcement.

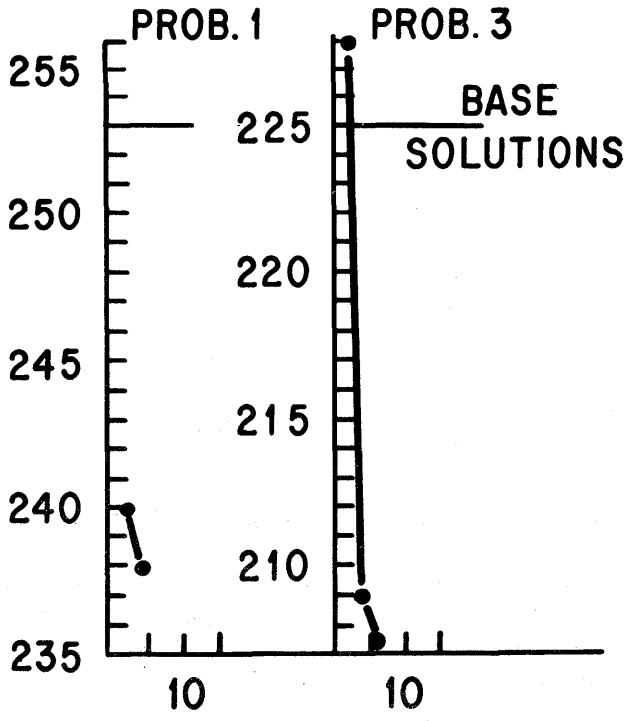


Figure 4.3—The solutions produced during test trials on Problems 1 and 3.

reinforcements. The improvement generalized to problems other than the training problem.

Figures 4.1 to 4.3 and 6.1 to 6.2 contain learning curves showing the improvement. The *base solution* to a problem is the average solution produced by random strategy selection. The improvement can be measured quantitatively by the ten-minute solution rate. The rates are shown in Table 3. A Mann-Whitney U test confirmed a highly significant difference ($p < .0001$) in rates on training and test trials.

Each segment of the learning curves in Figures 4.1 and 6.1 represents the performance from creation of an initial bench mark by using the highest valued state-strategy pairings until the program has not improved the bench mark in a specified time period. The bench mark is erased before a new segment starts. These segments are called training trials, but within any one of them there are many comparisons of solutions which may result in reinforcements. About 94% of all comparisons were indeterminate, i.e., neither the bench mark nor current solution was preferred. On the average 9 different state-strategy pairs were evaluated in a determinate comparison. One inevitable characteristic of an ordinal feedback system is that as learning progresses within a trial, positive comparisons become less frequent, and negative comparisons become more frequent. Altogether, during the three training trials on Problem 1, there were 449 positive reinforcements of state-strategy pairs, while, during the training on Problem 3, there were 142 positive reinforcements of state-strategy pairs. The change in one individual row in the table of connections is displayed in Figure 5.1. One can see that an equilibrium was reached early in the trial. The changes in the table of connections can also be measured in terms of the entropy of the table

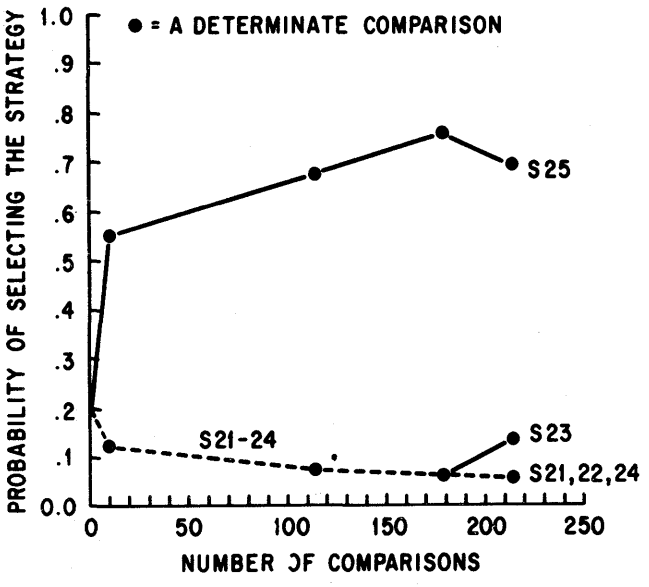


Figure 5.1—The probabilities of the program selecting each of the five strategies as a function of the number of comparisons during learning. The data are for one particular task situation (state) corresponding to average time requirements, average criticality, and the beginning of a solution. The data are from the first training trial on Problem 1 in Experiment I. The strategies are described in Table 2.

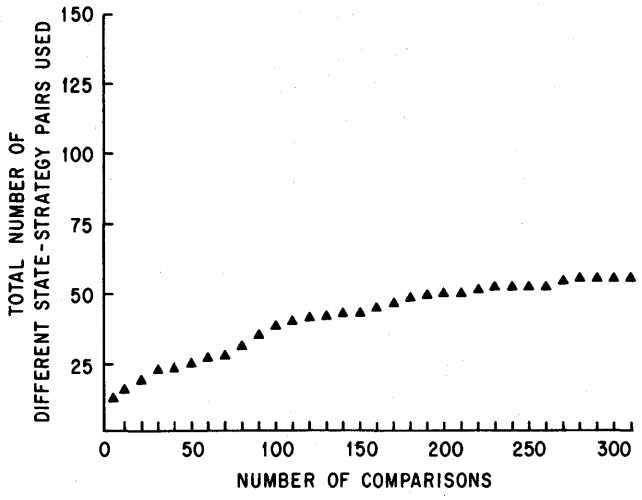


Figure 5.2—The number of different state-strategy pairings used as a function of the number of comparisons of partial solutions during learning. The data are from the first training trial on Problem 1 in Experiment I.

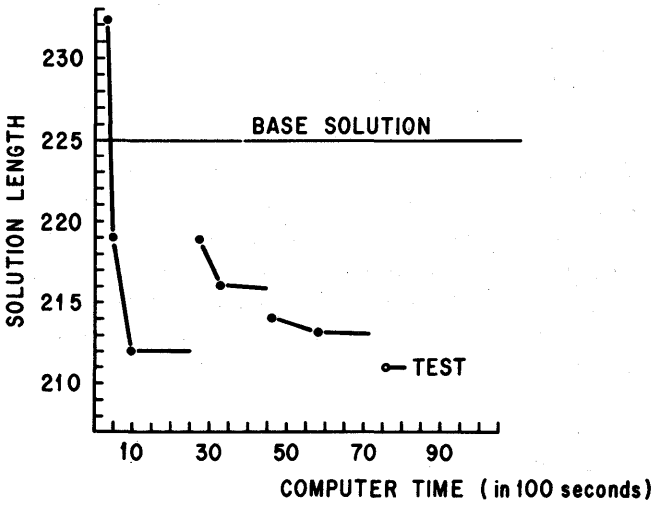


Figure 6.1—The solutions produced during training on Problem 3 with positive reinforcement.

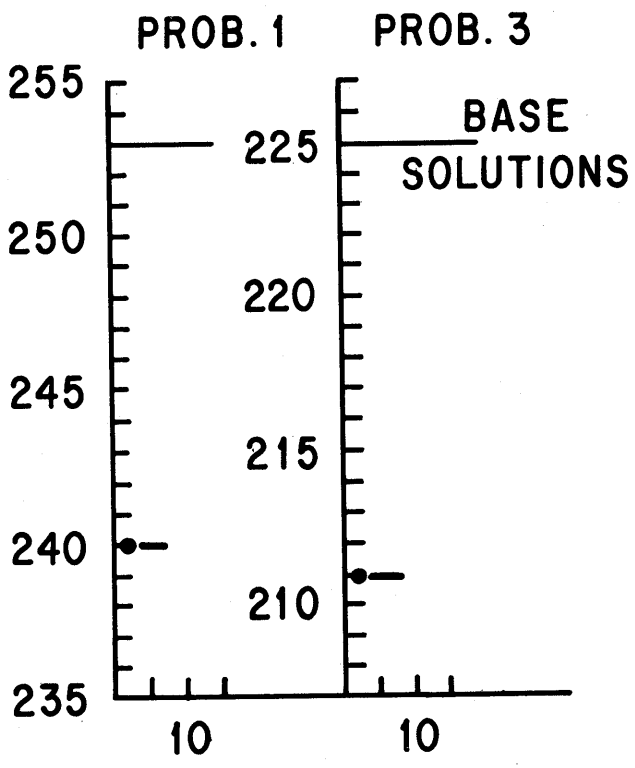


Figure 6.2—The solutions produced during test trials on Problems 1 and 2.

of connections. The entropy of the table trained on Problem 1 was reduced from 83.6 bits to 71.7 bits during training. This was 93% of the maximum possible reduction for the number of reinforcements. The final table of connections for training on Problem 1 is summarized in Table 4.

On the basis of relatively brief exposure to one optimization problem the performance program's table of connections was changed so that the program produced good solutions significantly more rapidly. This learning generalized to two other problems of the same type. Training from a naive state on these problems in turn improved performance on the original problem. Hence, significant learning is possible on a very complex optimization task with ordinal feedback and a state-vector representation of a reduced task environment.

EXPERIMENT II

(c) Does improvement occur more rapidly if the program changes its structure following its failures to improve its performance as well as after its success?

Having demonstrated that a learning mechanism based on ordinal feedback and using a state-vector representation will work, we can turn to the central issue in this study: should negative reinforcement (or error correction training) be used in a learning mechanism for optimization tasks?

The large majority of trainable pattern classifiers and game playing programs have used error correction training alone or in conjunction with positive reinforcement. This is somewhat surprising since the weight of evidence from psychology seems to indicate that positive reinforcement plays the most important role in learning while negative reinforcement may speed learning slightly by eliminating incorrect responses or may not help at all. Furthermore, we assert that error correction training is useful only if the learning program receives feedback data on an interval or ratio scale; feedback on an ordinal scale, as one receives in optimization tasks, while sufficient for positive reinforcement, is not sufficient for negative reinforcement (error correction training). In fact, error correction training or negative reinforcement should adversely affect the

TABLE III—Solution Rates During Experiment I (Positive Reinforcement)

First training series (Run 1)

	Trial	*Final Rate*	*Ten Minute Rate*
Training on Problem 1:			
	1	0.9	1.7
	2	1.2	1.3
	3	2.9	4.3
Additional training on Problem 2:			
	1	2.4	2.4
	2	2.9	2.9
Test on Problem 1:			
	1		5.0
Test on Problem 3:			
	1		5.1

Second training series (Run 2)

	Trial	*Final Rate*	*Ten Minute Rate*
Training on Problem 3:			
	1	2.9	1.7
	2	2.2	1.7
	3	1.5	3.1
Test on Problem 1:			
	1		4.3
Test on Problem 3:			
	1		4.0

TABLE IV—A Summary of the Table of Connections in Experiment I
(Both the mean success values and the probabilities of selection are given in each cell.)

FEATURE		*STRATEGIES*					
		S21	*S22*	*S23*	*S24*	*S25*	*SUM*
DEPTH IN SOLUTION	BEGINNING	11.8 .133	12.8 .145	13.0 .147	17.8 .201	33.0 .373	88.4
	MIDDLE	10.5 .108	21.5 .222	32.3 .333	17.0 .175	15.8 .162	97.1
	END	16.8 .218	11.3 .147	23.8 .309	10.5 .137	14.5 .189	76.9
TIME NEEDS OF EXECUTABLE JOBS	BELOW AVERAGE	12.8 .221	10.0 .173	10.0 .173	10.0 .173	15.0 .257	57.8
	ABOUT AVERAGE	16.3 .117	23.0 .165	43.3 .311	21.3 .153	35.3 .254	139.2
	ABOVE AVERAGE	10.0 .154	12.5 .193	15.8 .244	13.5 .208	13.0 .199	64.8
	$\leqq 1$	16.7 .192	18.7 .215	21.0 .242	11.7 .135	18.7 .215	86.8
VARIANCE OF CRITICALITY	BELOW AVERAGE	12.3 .138	16.7 .188	27.7 .311	19.7 .221	12.6 .142	89.0
	ABOUT AVERAGE	11.0 .120	14.0 .153	15.0 .164	11.0 .120	40.3 .441	91.3
	ABOVE AVERAGE	12.0 .146	11.3 .137	28.3 .344	18.0 .219	12.7 .154	82.3
MEANS		13.0 .149	15.2 .173	23.0 .263	15.1 .173	21.1 .242	

Strategies:
S21: Schedule job with minimum resource demands
S22: Schedule job with maximum time demand
S23: Schedule job with maximum criticality
S24: Schedule job whose time demand is closest to the remaining time for a scheduled job
S25: Schedule job with maximum resource demands

learning ability of a program trying to learn an optimization task.

Before showing why error correction training should hamper this type of learning, we need to review three key features of our ordinal learning program.

(a) The program possesses a preference routine that enables it to compare parts of new solutions with a bench mark solution.

(b) The program implements positive (or negative) reinforcement by incrementing (or decrementing) those cells in the table of connections (by Cpos or Cneg) that contributed to the new solution.

(c) The program uses strategy j in state i with a probability equal to $v_{ij} / \sum_{k=1}^{m} v_{ik}$ where v_{xy} is the value of the cell corresponding to state x and strategy y. Hence, the summation is over all strategies in state i.

Now we can state the theorem leading to our conclusion that error correction training will fail for any sequential

optimization task representable as finding the optimal path through a tree of feasible solution.

Theorem: For any ordinal-feedback learning procedure possessing characteristics a, b, and c, error correction training will *decrease* the probability of selecting the "best" strategy or response in each stimulus situation as soon as

(1) The "best" strategy is being used in over 50% of the situations encountered
(2) The probability that the bench mark will be preferred to a new solution is greater than Cpos/(Cpos + Cneg).

In less formal terms, when the bench mark solution and the table of connections have both become pretty good, negative reinforcement will begin to make the table of connections worse. Though this theorem will be proved for a program with characteristics a, b, and c, the reader should realize that the theorem (in slightly different form) will hold for viable alternatives to characteristics b and c. The central problem is that ordinal feedback becomes unreliable as the bench mark improves.

Proof:

Let Cpos be the increment for positive reinforcement.
Cneg be the decrement for negative reinforcement.
P be the probability that the bench mark solution is preferred after a determinate comparsion.
$V_t(i, j)$ be the entry in the table of connections corresponding to the ith state and the jth strategy (at time t).
$E_t(V)$ be the average value of $V_t(i, j)$ over all state-strategy pairs *used in constructing* a new path.
q be the % of situations in which "best" strategies were used in constructing the new path (% of situations for which "best" strategies exist in which they were used).

Now we can rewrite the two premises in the theorem as

(P.1) $q > .50$

(P.2) $P >$ Cpos/(Cpos + Cneg), Cpos > 0, Cneg > 0

From our description of the learning mechanism, we know that after a positive comparison

$$E_{t+1}(V) = E_t(V) + \text{Cpos} \tag{2.1}$$

and after a negative comparison

$$E_{t+1}(V) = E_t(V) - \text{Cneg} \tag{2.2}$$

Hence, the overall expectation following a determinate comparison is

$$E_{t+1}(V) = P^*(E_t(V) - \text{Cneg}) + (1 - P)^*(E_t(V) + \text{Cpos})$$

$$E_{t+1}(V) = E_t(V) + \text{Cpos} - (\text{Cpos} + \text{Cneg})^*P \tag{2.3}$$

but from P.2 we know $P >$ Cpos/(Cpos + Cneg); therefore

$$(\text{Cpos} + \text{Cneg})^*P > \text{Cpos}$$

and from (2.3),

$$E_{t+1}(V) < E_t(V)$$

In other words, once $P >$ Cpos/(Cpos + Cneg) we can expect the pairs used in constructing new paths to be decremented. As a result those pairs not used on the path will become more likely to be selected.

Let D be the expected decrement in the probability of selecting a "best" strategy that *was* used on the new path.
I be the expected increment in the probability of selecting a "best" strategy that *was not* used on the new path.

Since the probabilities of selection in any state must sum to unity, and since there are more than two strategies per state, and only one is decremented,

$$D > I \tag{2.4}$$

Now we can write an expression for the expected change in the probability of selecting a best strategy.

Let Δ_{prob} be the expected increase in the probability of selecting a "best" strategy after a reinforcement.

$$\Delta_{\text{prob}} = -q^*D + (1 - q)^*I \tag{2.5}$$

but from P.2,

$$q > .50 \qquad (q < 1)$$

$$q > (1 - q)$$

Therefore, using (2.4), we get from (2.5) that

$$\Delta_{\text{prob}} < 0$$

Hence, we have shown that the probability of selecting a "best" strategy must decrease, and our theorem is proved.

To test this hypothesis we attempted to train the program again from scratch on the same problems using both positive and negative reinforcement (the decrement for negative reinforcement was 1). The procedure was the same as in Experiment I, but the results were quite different.

TABLE V—A Comparison of Solution Rates in Experiments I and II

	(1) Mean 10 Minute Rate on First Two Learning Trials	(2)[a] Mean 10 Minute Rate on Test Trials	(2) − (1)
Experiment I (Positive Reinforcement)	1.6	4.0	+2.2**
Experiment II (Positive and Negative Reinforcement)	1.9	1.5	−0.4
Difference (I) − (II)	−0.3	+2.5*	

* $t = 3.309$, $df = 10$, $p < .005$
** $U = 0$, $p < .001$

[a] Including additional training on Problem 2.

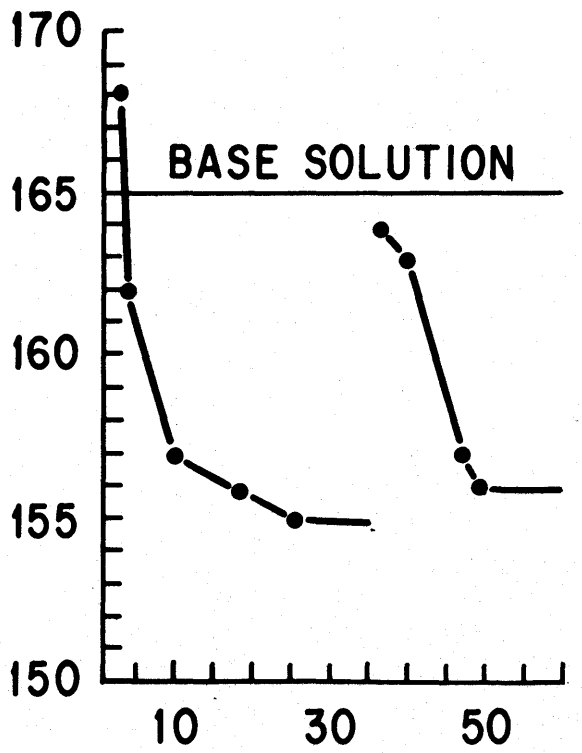


Figure 7.2—The solutions produced during additional training on Problem 2 with positive and negative reinforcement.

Results

The improvement in performance was significantly less for training with both positive and negative reinforcement than it had been in Experiment I for positive reinforcement alone.

In Table 5 the solution rates for training and test trials in Experiments I and II are compared. One can see that the solution rates were significantly inferior when negative reinforcement was included. This is also quite clear from the learning curves shown in Figures 7.1 to 7.3 and 9.1 to 9.2. The performance seems to have improved on the first training trial and then worsened. One can compare the test trials in this experiment with those in the first experiment (Figures 4.1 to 6.2) and note the substantial differences.

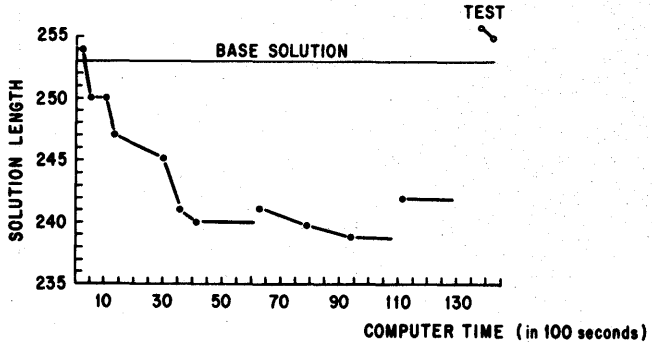


Figure 7.1—The solutions produced during training on Problem 1 with positive and negative reinforcement.

The reduction in entropy in the tables of connections trained with negative reinforcement was also less. The reduction in the table trained on Problem 1 was only 13% of the possible. One should not overemphasize this difference, however, since orderliness does not necessarily imply that the desired order has been achieved. Nevertheless, it is interesting to note that 90% of the 13% reduction in entropy during training on Problem 1 occurred during the first training trial. This, of course, is in accord with our hypothesis.

One can see some of the more subtle effects of negative reinforcement more clearly by looking at the within trial behavior of the program. During the three training trials on Problem 1 and the following two trials on Problem 2 there were 518 positive reinforcements of state-strategy pairs in contrast to 1701 negative reinforcements of strategy pairs. Let us look in detail at the behavior of the program within Training Trial 1 on Problem 1 and compare it with the corresponding training trial in Experiment I. The rate of occurrence of negative comparisons was slightly less in this experiment. This is not surprising since negative reinforcements would make it less likely that the program would repeat a series of bad decisions and encounter

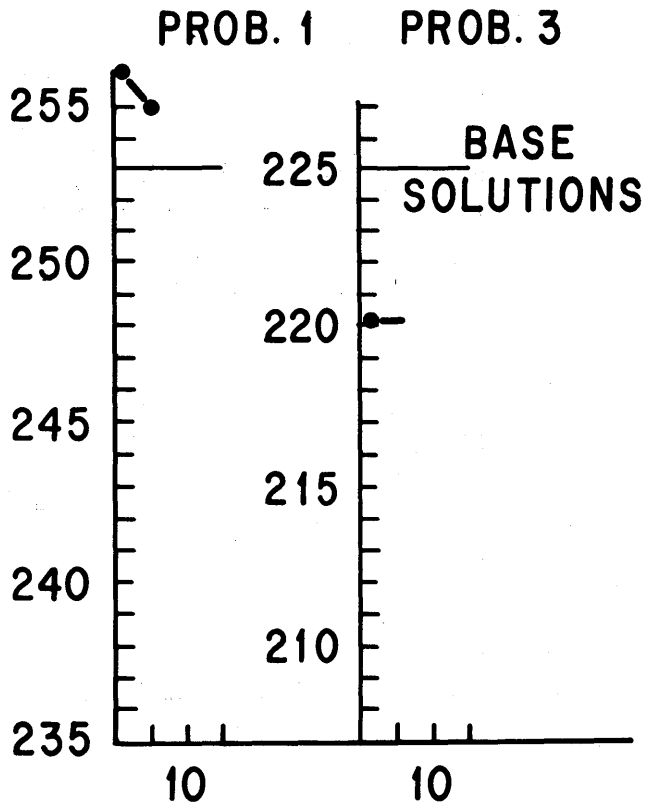


Figure 7.3—The solutions produced during test trials on Problems 1 and 3.

another negative reinforcement. From the large fluctuations in the sum of the entries in the table of connections during the trial, we could see that positive reinforcements were the dominant influence *at the beginning* of the trial, but that their effect could have been wiped out by the negative reinforcements later in the trial. Besides preventing the repetition of bad decision sequences, negative reinforcement introduces more variety into the decision making process. In other words, negative reinforcement can move the table of connections off a locally optimal structure to search for a better structure. The greater variety in decision making is best seen by comparing Figure 8.2 with Figure 5.2. However, the total effect of these characteristics of negative reinforcement in changing the structure of the table of connections is best shown by Figure 8.1. With negative reinforcement the changes in probability were more erratic. A new strategy suddenly increased in probability after the trial was half over.

This experiment demonstrates that negative reinforcement can never be used as freely as positive reinforcement in learning optimization tasks. The many previous methods based solely on error correction train-

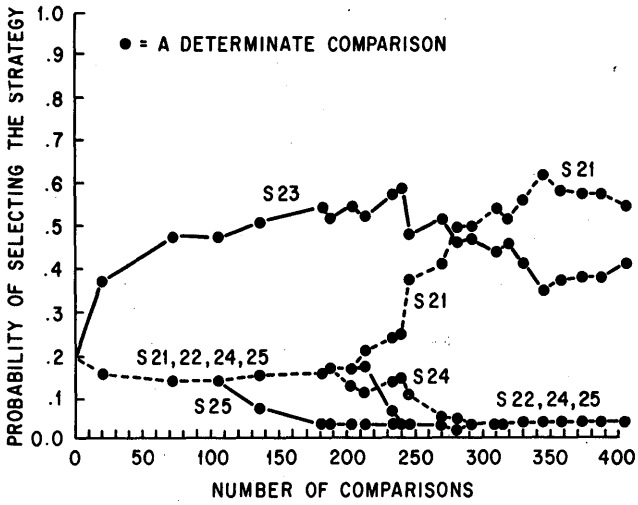


Figure 8.1—The probabilities of the program selecting each of the five strategies as a function of the number of comparisons of partial solutions during learning. The data are for one particular task situation (state) corresponding to average time requirements, average criticality, and the beginning of a solution. The data are from the first training trial on Problem 1 in Experiment II.

ing would perform poorly on optimization tasks. Nevertheless, one can see that negative reinforcement has some desirable effects: it prevents the table of connections from becoming stranded on local optima and causes a greater variety of decisions to be investigated.

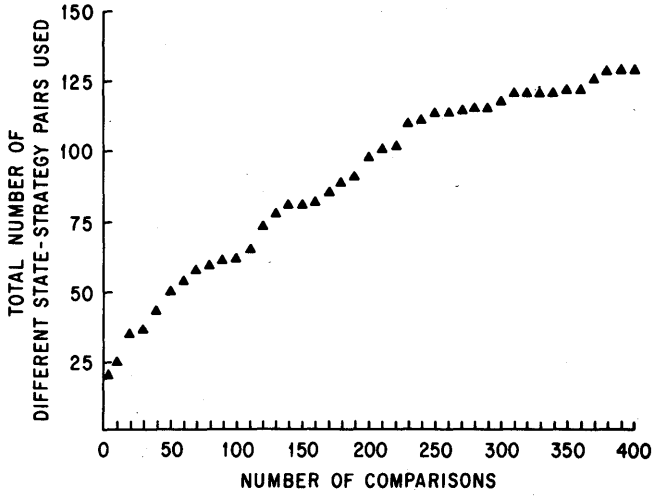


Figure 8.2—The number of different state-strategy pairings used as a function of the number of comparisons of partial solutions during learning. The data are from the first training trial on Problem 1 in Experiment II.

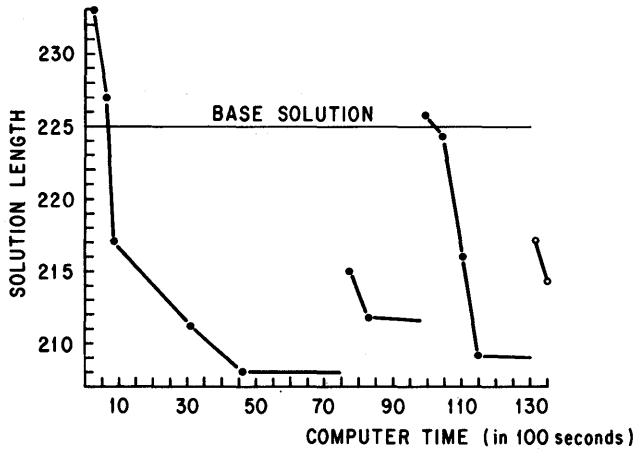


Figure 9.1—The solutions produced during training on Problem 3 with positive and negative reinforcement.

Hence, we would like to find a way to reap some of the benefits of negative reinforcement while avoiding the pitfalls.

EXPERIMENT III

(d) During training should the program always strive to produce the best possible solution?

An implicit assumption in many previous learning mechanisms is that the path to becoming an excellent problem solver is monotonic. However, if this assumption—that the strategies employed by a good problem solver will be worthwhile for an expert—is not always true, then a learning program needs a mechanism for exploring solutions outside of those suggested by its previous learning? In different terms, doesn't a learning program need a way to escape local optimums generated by particular strategies and to experiment with new strategies that may lead to the global optimum?

If one views an optimization problem as the problem of finding an ideal path in a tree of solutions, he can see why a learning program would need such mechanisms. The strategies that generate a path (solution) are reinforced if that path is superior to previous paths. Desirably enough, this somewhat narrows the scope of future search to branches likely to be selected by the same strategies. Eventually, a solution will be reached that cannot be exceeded in a reasonable amount of time. At this point all branches off this path (assuming some entropy in the search process) and all branches likely to be reached with the same strategies should have been tried and found inferior. But there is no guarantee that a radical change in several strategies at some point on the path might not lead to an equal or better path. The danger is that a few radical changes in strategies might consistently produce quite different paths and superior solutions, but these changes would never be investigated because any one of them, alone, coupled with the learned series of strategies, only leads to a branch off the old path and a worse solution. Therefore, it is suggested that a program that adopts a short period of relatively non-directive search at the end of a learning sequence where improvement has terminated will learn a superior decision structure and eventually perform better than a program that spends all its time searching on the basis of its past experience.

Admittedly, such a non-directive search would be time consuming and costly in that performance would be bad during learning. MacKay,[8] in fact, has suggested deliberately selecting bad strategies during learning so that they can be eliminated with negative reinforcement. However, such a method would not help much in selecting a strategy of little utility most of the time that is of the highest utility in moving from good to excellent solutions. It is suggested that what is needed is a mechanism for relatively random exploration of strategies whenever it appears that the program is "hung up" on a local optimum. How useful such an addition to a learning procedure would be is the fourth issue for study.

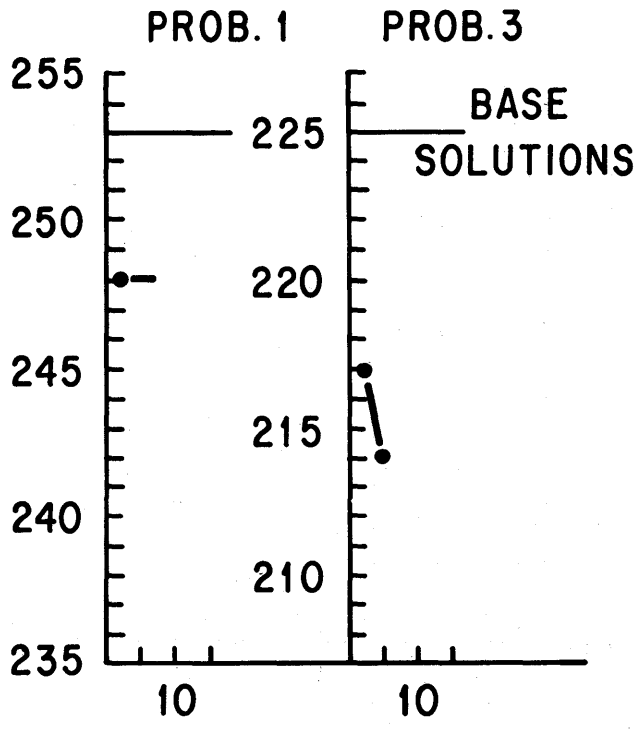


Figure 9.2—The solutions produced during test trials on Problems 1 and 3.

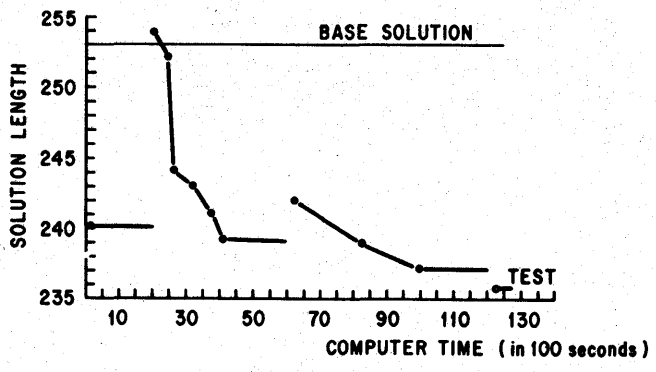


Figure 10.1—The solutions formed during special additional training on Problem 1 in Experiment III. The initial memory structure has been trained with positive reinforcement alone in Experiment I (see Figs. 4.1 and 4.2). Both positive and negative reinforcement were used on these three trials. In addition, after the first bench mark was formed, startegies were selected completely at random during Trials 1 and 3.

We began with the final table of connections from training on Problem 1 in Experiment 1. It appeared that this table had become stranded on a local optimum. Although it produced good solutions, better solutions existed that it could not find. Additional training did not help since improvement is needed for positive

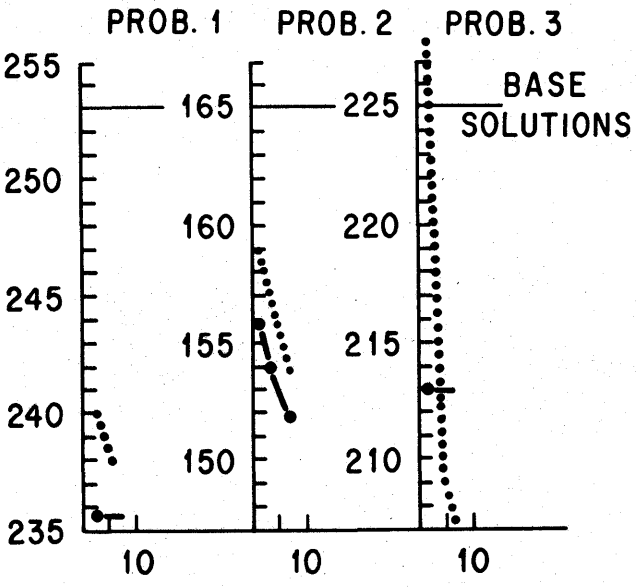


Figure 10.2—The solutions produced during test trials on Problems 1, 2, and 3. The broken lines represent the solutions formed during the old test trials before this additional training.

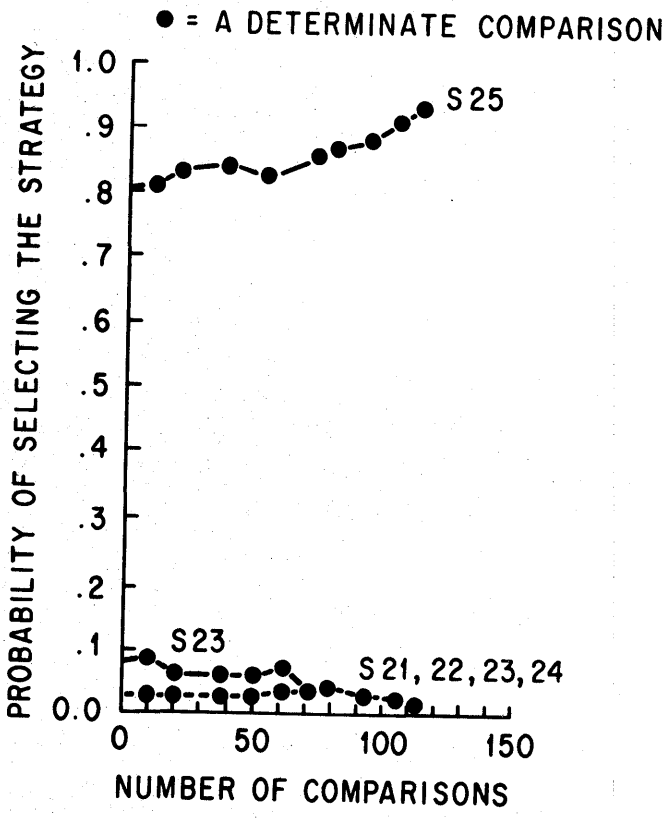


Figure 11.1—The probabilities of the program selecting each of the five strategies as a function of the number of comparisons of partial solutions during learning. The data are for the state corresponding to average time requirements, average criticality, and the beginning of a solution. The data are from the first training trial on Problem 1 in Experiment III, a trial during which strategies were selected at random.

reinforcement to be applied. Hence, we decided to give the table of connections additional training with negative reinforcement *and with strategies selected at random.* With this scheme good strategies would be no more likely to be selected than bad ones, and negative reinforcement should not destroy the table of connections; rather it should move the table off the local optimum and allow the learning program to search for another optimum.

Results

The results of the experiment indicate that this is exactly what happened. Two random strategy selection trials were combined with one normal training trial. These are shown in Figure 10.1. During the first trial,

TABLE VI—A Summary of the Table of Connections in Experiment 3
(Both the mean success values and the probabilities of selection are given in each cell.)

FEATURE		*STRATEGIES*				
		S21	*S22*	*S23*	*S24*	*S25*
DEPTH IN SOLUTION	BEGINNING	6.58 .084	8.08 .103	7.25 .093	16.33 .208	40.00 .511
	MIDDLE	7.17 .076	18.17 .192	45.33 .479	13.92 .147	9.92 .105
	END	18.58 .226	9.92 .120	30.25 .368	10.08 .122	13.25 .161
TIME NEEDS OF EXECUTABLE JOBS	BELOW AVERAGE	12.92 .241	8.58 .160	8.42 .157	10.17 .190	13.58 .253
	ABOUT AVERAGE	10.50 .072	18.83 .129	56.67 .389	20.83 .142	39.67 .271
	ABOVE AVERAGE	8.92 .138	8.75 .135	17.75 .274	9.33 .299	9.92 .153
VARIANCE OF CRITICALITY	≤ 1	17.56 .203	19.44 .225	21.44 .248	11.56 .134	16.33 .189
	BELOW AVERAGE	8.56 .101	12.56 .148	38.23 .450	18.56 .219	7.00 .082
	ABOUT AVERAGE	8.33 .087	10.00 .104	14.56 .152	9.11 .949	54.00 .563
	ABOVE AVERAGE	8.67 .119	6.23 .086	36.23 .499	14.56 .201	6.89 .095
MEANS		10.78 .127	12.06 .142	27.61 .325	13.44 .158	21.06 .248

Strategies:

S21: Schedule job with minimum resource demands
S22: Scheule job with maximum time demand
S23: Schedule job with maximum criticality
S24: Schedule job whose time demand is closest to the remaining time for a scheduled job
S25: Schedule job with maximum resource demands

where strategies were chosen at random, negative reinforcement altered the table of connections moving it off its local peak. The second trial, a normal training trial, established new connections in the table with positive reinforcement. The third trial again varied the table with negative reinforcement and random strategy selection. Finally, on a test trial (Figure 10.2) the performance program produced good solutions to all three problems and the best solutions to Problems 1 and 2 that were ever generated. The final table of connections is displayed in Table 6.

A study of the within trial behavior of the program

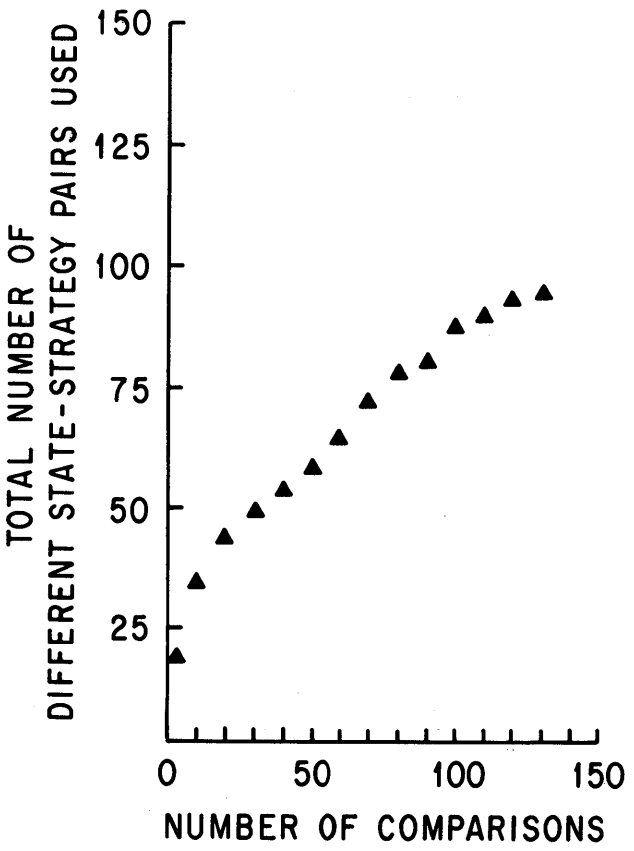


Figure 11.2—The number of different state-strategy pairings used as a function of the number of comparisons of partial solutions during learning. The data are from the first training trial on Problem 1 in Experiment III, a trial during which strategies were selected at random.

confirms these effects. Comparing Figure 11.2 with Figures 5.2 and 8.2 indicates that a greater number of state-strategy pairings were investigated in this learning mode than in the other modes. The average value of cells in the table of connections dropped rapidly, making the table more susceptible to new applications of positive reinforcement. At the same time this random selection mode did not destroy all the previous learning. The random selection trial actually increased the superiority of strategy S25 in the state shown (Figure 11.1).

From these results one can conclude that it is not always the best policy during learning to try and find good solutions. At some times one is better off investigating new state-strategy pairings and ignoring the immediate consequences.

CONCLUSIONS

The study demonstrated that a program for solving a very complex optimization task (a scheduling task) could learn generalizable performance principles solely through ordinal feedback from intercomparisons of the solutions it had produced. In particular connections were formed from a very restricted state description of the environment to basic performance strategies. Only a relatively small number of positive reinforcements were necessary to produce significant improvement. Furthermore, and perhaps most important of all, negative reinforcement or error correction training was shown to be a hindrance to learning when only ordinal feedback was available. Theoretical evidence was provided in support of this empirical finding. Finally, the study established that some benefits can be derived from negative reinforcement by using it with a performance routine that does not always preform at its best.

This study did not address (directly) the problem of how the program could generate the stimulus features and basic strategies that are the entries in the table of connections. However, a table of connections provides a great deal of information that can be used in eliminating old entries and introducing new entries. Some simple descriptive statistics about a table, e.g., variance of subsets of entries, provide information on just what basic strategies or features should be eliminated. Several algorithms have already been developed to make use of this information.

Finally, this study raises some very interesting questions about human learning ability. Do humans sometimes ignore the consequence of their behavior in order to learn more rapidly? What is the relative power of ordinal, interval, and ratio feedback in learning complex tasks? Do humans change their state-strategy pairings on the basis of failures to improve? How large a set of state variables do humans attend to at any one time?

REFERENCES

1 M MINSKY S PAPERT
Perceptrons
MIT Press Cambridge Massachusetts 1969

2 N J NILSSON
Learning machines
McGraw-Hill New York 1965

3 D B YNTEMA G E MUESER
Keeping track of variables that have few or many states
Journal of Experimental Psychology Vol 63 No 4 pp 391–395 1962

4 H FISHER G L THOMPSON
Probabilistic learning combinations of local job-shop scheduling rules
Industrial Scheduling (eds M Muth and G L Thompson)
Prentice-Hall Englewood Cliffs 1963

5 A NEWELL J C SHAW H A SIMON
A variety of intelligent learning in general problem solver
Self-Organizing Systems (eds M Yovitts and S Cameron)

Pergamon Press 1960
6 A L SAMUEL
Some studies in machine learning using the game of checkers
IBM Journal of Research and Development pp 211–229
July 1959
7 A L SAMUEL
Some studies in machine learning using the game of checkers
II—Recent progress
IBM Journal of Research and Development Vol 11 No 6
pp 601–617 November 1967
8 D MACKAY
Information and learning
Lernende Automatica (ed H Billing) Munchen Oldenbourg
1961

Man-machine interaction for the discovery of high-level patterns

by DAVID F. FOSTER*

General Electric Company
Bethesda, Maryland

Social scientists are largely concerned with the discovery of patterns and relationships in multivariate data. The techniques used have been, for the most part, the standard tools of multivariate analysis—correlation, regression, factor analysis, and so forth. Unfortunately, the nature of these techniques has tended to impose implicit theoretical assumptions and constraints on the social sciences. The conceptualization of a variable as the weighted sum of other variables is methodologically unhealthy in the study of human beings and societies, in which it is precisely the complex interaction effects among variables which are of the most theoretical interest. In the social sciences it is especially prevalent, and especially significant, for the relationship between two variables to be dependent on the values of other variables—and for the nature of this dependence to be a function of still *other* variables. Traditional statistical methods are ill-suited for uncovering such hierarchies of interaction effects.

Progress in this direction has been made through the pattern-analytic methods developed by Dr. Louis McQuitty[1,2] and others. These methods develop classifications of individuals by searching for clusters of variables on which groups of individuals tend to agree. Another approach has been taken by researchers at System Development Corporation with their time-sharing programs TRACE and IDEA.[3,4] These programs make it possible for a social scientist to interact on-line with his data-base, making possible the discovery of interactions which would be almost impossible to discover through batch processing and standard techniques of statistical analysis.* The use of time-sharing is especially significant here. Searching for complex patterns is a process which inherently involves enormous amounts of combinatorial manipulation. Through time-sharing, a researcher in continuous interaction with a computer can use his theoretical knowledge and common sense to greatly reduce the amount of effort that would have to be expended by a "blind" program. This is a significant example of the "augmentation of human intelligence" which has been talked about so much in connection with time-sharing. The researcher can explore his data heuristically, developing hypotheses and testing them as he goes along. This kind of interaction will permit the development of theoretical concepts which are more complex and more interesting than many of those now existing in the social sciences.

HIGH-LEVEL PATTERNS

In this paper, I wish to introduce the concept of a *high-level pattern*, and to propose that while the discovery of such patterns in social science data is usually beyond the reach of either an unaided human being or an unaided computer, it can be accomplished by a symbiosis of the two. A high-level pattern is different from the kind of pattern sought by the pattern-analytic methods of McQuitty and others. Whereas they seek patterns which are partitions of a set of individuals into subsets of individuals who agree among themselves on certain groups of variables, a high-level pattern is a partition of the set of individuals into subsets *within which a common set of linear, additive, non-interactive relationships holds.* That is, although there may be no significant correlations among variables for a certain sample of individuals, it is possible that there exists a partition of the sample into subsets within each of which there exist very strong correlations between variables.

The existence and discovery of such partitions is most significant from a theoretical viewpoint in the social sciences. If it is true, as Robinson and I have

* The opinions stated in this paper are the responsibility of the author alone.

* TRACE and IDEA have been applied to numerous investigations, including an interesting study of the Watts riot. In this study it was found that the effect of age on participation in the riot was directly opposite for men and women—younger men participated, while older women did. This is an example of the kind of effect which would not show up in a correlation or regression analysis.

suggested in another paper,[5] and as Peter Winch has cogently argued,[6] that the structures of societies are best regarded not as something external, like physical laws, but as reflections of the ways in which individuals organize experiences into conceptual frameworks, then one might expect to find in a set of social data not one set of relationships, but *multiple* sets of relationships, partially separate and partially overlapping. In a study of political beliefs, for example, it might be found that there exists a subset of people whose beliefs can best be understood in terms of a conflict between the economic haves and have-nots. The attitudes of people within this subset could differ completely; they could include both John Birchers and Marxists, but they would be classified together in a high-level pattern because their beliefs could be explained in terms of a *common correlation matrix*, i.e., their beliefs are not the same, but the pattern of *relationships* among their various attitudes is. Another group might consist of people whose alignment is basically in terms of the dimension "authoritarianism" vs. "permissiveness," and this group would have a separate correlation matrix for the attitude variables. Then there would probably be a group of people whose attitude configurations could be understood in terms of *either* correlation matrix, and so on. It can be seen that the breakdown of a sample into types within which different relationships among variables hold would be of the greatest interest and theoretical importance to social scientists. However, the discovery of such types would be practically impossible for an unaided computer program—the combinatorial problem in trying all possible types would be too vast, even with clever heuristics, and there is a strong possibility that the typology found would be spurious—due largely to sampling variation and theoretically meaningless. The discovery of a high-level typology by an unaided individual is also practically out of the question—it is difficult enough to discover simple interrelationships in data, let alone this type of highly abstract pattern. Man-machine interaction can provide a possible answer.

A TIME-SHARING PROGRAM FOR DISCOVERING HIGH-LEVEL PATTERNS

As a step toward implementing this concept, a program has been written (for the GE time-sharing system) to carry out an interactive search for patterns of this type. Although primitive, the program provides an indication of the sort of thing that can be done in this area. The program provides the researcher with a set of simple commands, which are to be used heuristically—he can try arranging individuals in one way, see what happens, try another, and so on. Some of the available commands are given below:

KEY,n and FILE,n will cause data records for n individuals to be read into memory from the teletype or the disc file, respectively.

INC,j and EXC,j cause individual j to be included or excluded "current subset"—the subset of individuals under consideration at any given time.

CORR causes the complete intervariable correlation matrix to be calculated for all the individuals in the current subset. The matrix is printed. 2×2 frequency distribution tables are calculated for each possible pair of variables (the present version of the program is limited to dichotomous data). The frequency tables are not printed because this would be far too time-consuming and unwieldy for an interactive program.

DIFF,i divides the current subset into two further subsets—all the individuals for which the value of variable i equals k_1, and those for which the value of variable i equals k_2, where k_1 and k_2 are the possible responses on each variable. Correlation matrices are calculated for both subsets, and

$$Q = \sum_{j=1}^{n} \sum_{m=j}^{n} (A_{1jm} - A_{2jm})^2, \quad m \neq j \neq i$$

is calculated and printed, where A_{1jm} is the correlation matrix for the subset in which the value of variable $i = k_1$ and A_{2jm} is the matrix for the subset in which the value of $i = k_2$. This provides an index of the degree to which variable i affects the total structure of the data. If Q is large, it is a sign that variable i distinguishes between groups of individuals within which different sets of relationships hold among the variables.

SEL,i,j takes the current subset and excludes from it all individuals for which the value of variable i is not equal to k_j. That is, it partitions the current subset on a selected variable. SEL, DIF, and CORR can be used together to effectively investigate hierarchies of interaction effects.

RANK provides an indication of the degree to which individual in the sample "belongs" to the high-level type represented by the individuals comprising the current subset. That is, it indicates the degree to which each individual "fits" pattern of interrelationships between variables holding within the current subset. This is done by looking at the 2×2 frequency tables calculated by CORR for each variable pair and summing the probabilities of occurrence for each combination actually occurring for an individual. This is not a particularly sophisticated estimator, statistically but it does

provide a heuristically useful gauge of the degree of "belongingless" of each individual to a given high-level type.

There are in addition other commands, such as FORCE, which permits one to ignore a particular relationship in a type if one suspects that it is spurious. It is easy to think of other commands which would be useful in investigating the structure of the data and uncovering high-level types. One particular set of commands, which will be implemented soon, will permit the user to give a name to a subset of individuals or variables and make it current at a later time by merely saying GET,*xxx*, where *xxx* is the name. Also planned are optimizing routines which will heuristically attempt to build the most cohesive possible high-level types around a few individuals selected by the researcher to represent a hypothetical type, within constraints imposed by the researcher. (For example, the correlation between variables 2 and 7 must be positive.) This latter command is likely to cause problems because of excessive use of CPU time.

This program, which is called INTERFORM, for INTERactive pattern FORMation, has as yet been tested only on made-up data, not real-world social science data. It is planned to eventually embed it in a general-purpose program for the analysis of social science data, along the lines of SDC's TRACE.

COMMENTS

This kind of interactive data manipulation is viewed with distaste by many statisticians—perhaps justifiably so, when one considers that you can find any kind of relationship you are looking for if you subdivide the data enough different ways. It must be emphasized that one can never regard the results of this or any similar program or procedure as "proof" of anything—only as a way to develop interesting concepts and hypotheses.

The susceptibility of INTERFORM and similar programs to sampling fluctuations is, however, greatly reduced by the fact that it looks at *multiple* relationships—between each variable and each other variable—rather than just between variables and a single criterion variable. This would seem to be a useful idea for pattern-recognition programs in general; to search for relationships between the component parts of the pattern as well as between the pattern and the criterion variable. (The relationship between social science data analysis and artificial intelligence work has been too little noticed. Inasmuch as human beings are intelligent (?), it is reasonable to expect that the patterns formed in their interactions with one another might be of a kind and complexity similar to the problems dealt with in AI.)

The interactive procedures discussed here should provide new perspectives in many areas of the social sciences, especially in the study of anomie (social disorganization)[7] and the study of the formation of coalitions and issue alignments in politics. This kind of analysis is an example of the way in which man-machine interaction can be exploited not only to perform existing tasks better, but to do things which could perhaps otherwise not be done *at all*. Given the nature of social science data, the potentialities of time-sharing as an intelligence amplifier in the field are practically unlimited.

REFERENCES

1 L McQUITTY
The mutual development of some theories of types and some pattern analytic methods
Presidential Address Division V Evaluation and Measurement The American Psychological Association September 1965

2 L McQUITTY
Dr McQuitty's papers have appeared frequently in Education and Psychological Measurement for the last several years and the interested reader is referred to this source

3 L PRESS M ROGERS
IDEA—A conversational heuristic program for interactive data exploration and analysis
Proc of the 22nd National ACM Conference 1967

4 G SHURE R MEEKER W MOORE
TRACE: Time-shared routines for analysis, classification and evaluation
Spring Joint Computer Conference Proceedings 1967

5 I E ROBINSON D FOSTER
The simulation of human systems
Digest of the Second Conference on Applications of Simulation 1968

6 P F WINCH
The idea of a social science 1958

7 The concept of anomie was developed by French sociologist Emile Durkheim. An anomic individual is conceptualized as being alienated, unable to find meaning or structure in the organization of the world or society. It is reasonable to postulate that, within the theoretical framework developed in this paper, anomic individuals would lack clear membership in any high-order type—i.e., the variations in their beliefs would be unstructured and random.

Completeness results for *E*-resolution*

by ROBERT ANDERSON

University of Texas
Austin, Texas

INTRODUCTION

Since their introduction in 1965,[7] resolution based deductive systems for the first-order predicate calculus have been extensively investigated and utilized by researchers in the field of automatic theorem-proving by computer. Part of this research has been directed at finding techniques for treating the equality relation within the framework of resolution based deductive systems.[2,3,4,5,9,10] Perhaps the most natural treatment of equality, introduced so far, is by means of the paramodulation principle which when used in conjunction with resolution forms a complete deductive system for the first-order predicate calculus with equality.[5,6,11] A very similar technique for treating equality was introduced[4] and called *E*-resolution. In fact *E*-resolution can be viewed as a restricted form of paramodulation and resolution. The purpose of this paper is to define *E*-resolution in terms of paramodulation and resolution and to prove the completeness of *E*-resolution and several modifications of *E*-resolution.

PRELIMINARIES AND TERMINOLOGY

The reader is assumed to be familiar with the notation and terminology of resolution and paramodulation.[5,6,7,8,11] In addition the reader is assumed to be familiar with the technique introduced in Reference 1, for establishing the completeness of resolution based deductive systems. In that regard, recall that the technique is based on mathematical induction on the parameter k (the excess literal parameter) defined as follows: For any set S of clauses, $k(S)$ is defined to be (the total number of appearances of literals in S) minus (the number of clauses in S). To define *E*-resolution in terms of paramodulation and resolution we need the following definitions:

Definition 1. If S is a set of clauses and C is a clause in S and l is a literal in C then define

$P^{(0)}(S, C, l) = \{C\}$

$P^{(1)}(S, C, l)$ = the set of all clauses which can be obtained from C by paramodulating from some clause in S into the literal l in C.

and by induction,

$P^{(n)}(S, C, l)$ = the set of all clauses which can be obtained from clauses $C' \in P^{(n-1)} \cdot (S, C, l)$ by paramodulating from some clause in S into the literal l' (in C'), which is descended from the literal l in C.

Definition 2.

$$P^{(\infty)}(S, C, l) = \bigcup_{n \in \text{integers}} P^{(n)}(S, C, l).$$

Thus $P^{(\infty)}(S, C, l)$ consists of all the clauses which can be obtained from C by paramodulating any finite number of times from clauses in S into the literal l (or its descendents) in C. There is no paramodulation into any other literal in C.

Definition 3. If S is a set of clauses then C_3 is an *E*-resolvent of S iff there exist clauses C_1 and C_2 in S and literals l_1 in C_1 and l_2 in C_2, and there exist clauses $C_1' \in P^{(\infty)}(S, C_1, l_1)$ and $C_2' \in P^{(\infty)}(S, C_2, l_2)$ such that C_3 is a resolvent of C_1' and C_2' and the literals resolved upon in C_1' and C_2' are those descended from l_1 and l_2 respectively.

Thus *E*-resolution consists of paramodulation and resolution used in the following manner: Two clauses are selected and a literal is selected from each clause as possible literals to be resolved upon. One then searches (in a depth first manner) for all possible ways of uni-

* This work was supported by National Institute of Health Grant GM 15760-02.

fying the selected literals by means of paramodulating into them and the unifications algorithm. Thus the only paramodulation which is done is paramodulation directly into the selected literals (or their descendents). The idea being that this restricts paramodulation to those particular paramodulations which are immediately relevant. Of course in actual practice one cannot search through the countable number of stages involved in generating $P^{(\infty)}(S, C, l)$ and therefore one must utilize a cut-off parameter (called the tree level bound).[4] But by constantly increasing the cut-off parameter as one proceeds through a proof, any possible E-resolvent can still be generated. Also one does not actually need to make the substitutions involved in generating $P^{(\infty)}(S, C_1, l_1)$ and $P^{(\infty)}(S, C_2, l_2)$ unless and until a series of substitutions are found which do actually unify l_1 and l_2. Instead one may simply generate a tree of possible substitutions [see Reference 4 for details].

COMPLETENESS RESULTS FOR E-RESOLUTION

Theorem 1. (Ground Completeness of E-resolution) If S is an R-unsatisfiable set of ground clauses, and S' is the set of ground clauses obtained by adding to S all ground instances of the clause $(x = x)$, then $\square$ can be deduced from S' by E-resolution.

Proof. The proof follows the general outline given in theorem 1 of Reference 1 and is by induction on $k(S)$.

(i) Suppose $k(S) = 0$. Then S must consist entirely of unit clauses. Since paramodulation and resolution is known to be complete,[6] we know that $\square$ can be deduced from S' by paramodulation and resolution. But since S' consists entirely of unit clauses there can be only one resolution involved, since any resolvent of unit clauses is $\square$. Thus this deduction of $\square$ must consist of a series of (0 or more) paramodulations followed by a single resolution. It is clear that all relevant paramodulations can be made into the two literals which are finally resolved upon. Thus $\square$ is in fact an E-resolvent of S'.

(ii) The induction step $(k(S) = N)$ follows identically the outline given in theorem 1 of Reference 1 with the word "resolution" replaced by the word "E-resolution."

Wos and Robinson[11] extended the concept of set of support to cover paramodulation as well as resolution and showed the completeness of paramodulation and resolution with set of support. Since E-resolution is a restricted form of paramodulation and resolution, their definition of set of support can be immediately applied to E-resolution. Using this definition we obtain

Theorem 2. (Ground completeness for E-resolution with set of support) If S is an R-unsatisfiable set of ground clauses and S' is the set of ground clauses obtained by adding to S all ground instances of the clause $(x = x)$ and $T \subset S$ is such that $S' - T$ is R-satisfiable, then $\square$ can be deduced from S' by E-resolution with T as set of support.

Proof. The proof is again by induction on $k(S)$.

(i) For $k(S) = 0$ we know that S' consists entirely of unit clauses. Since paramodulation and resolution with set of support is complete,[11] we know that there must exist a deduction of $\square$ from S' by paramodulation and resolution with T as set of support. As in theorem 1 the fact that S' consists entirely of unit clauses assures us that any such proof can be given the particular form defined as E-resolution and thus we obtain a deduction of $\square$ by E-resolution with T as set of support.

(ii) The proof of the induction step $(k(S) = N)$ follows exactly the outline given in theorem 3 of Reference 1 where the word "resolution" is replaced by the word "E-resolution" and the word "satisfiable" is replaced by the word "R-satisfiable."

In order to extend the concept of hyper-resolution[8] to hyper-E-resolution we need the following technical definitions:

Definition 4. If S is a set of clauses and $C \in S$ and literals $l_1, \cdots, l_n$ occur in C then define

$P^{(0)}(S, C, \{l_1, \cdots, l_n\}) = \{C\}$

$P^{(1)}(S, C, \{l_1, \cdots, l_n\})$ = the set of all clauses which can be obtained from C by paramodulating from some clause in S into one of the literals $l_1, \cdots, l_n$ in C and where the clauses from which paramodulation occur consist entirely of positive equality literals.

and by induction,

$P^{(n)}(S, C, \{l_1, \cdots, l_n\})$ = the set of all clauses which can be obtained from clauses $C' \in P^{(n-1)}(S, C, \{l_1, \cdots, l_n\})$ by paramodulating from some clause in S into one of the literals $l_1', \cdots l_n'$ (in C') descended from $l_1, \cdots, l_n$, and where the clauses from which paramodulation occur consist entirely of positive equality literals.

Again we define,

$$P^{(\infty)}(S, C, \{l_1, \cdots, l_n\}) = \bigcup_{n \in \text{Integers}} P^{(n)}(S, C, \{l_1, \cdots, l_n\})$$

Definition 5. If S is a set of clauses then C is a hyper-E-resolvent of S iff there exist clauses $C_0, C_1, \cdots, C_n$ in S where

1) the only negative literals in C_0 are $\sim l_1, \cdots, \sim l_n$,
2) $C_1, \cdots, C_n$ are all positive clauses,
3) l_1' is in $C_1, \cdots$, and l_n' is in C_n and C is a hyper-resolvent of some clauses $C_0' \in P^{(\infty)}(S, C_0, \{\sim l_1, \cdots, \sim l_n\})$, $C_1' \in P^{(\infty)}(S, C_1, \{l_1'\}), \cdots, C_n' \in P^{(\infty)}(S, C_n, \{l_n'\})$, resolved upon the literals descended from $\sim l_1, \cdots, \sim l_n, l', \cdots, l_n'$.

Thus hyper-E-resolution is essentially hyper-resolution and paramodulation in which we restrict the paramodulation so that it occurs only into the literals to be resolved upon. In addition we require that paramodulation occur only from clauses consisting entirely of positive equality literals.

Theorem 3. (Ground completeness for hyper-E-resolution) If S is an R-unsatisfiable set of ground clauses and S' is the set of ground clauses obtained by adding to S all ground instances of the clause $(x = x)$, then $\square$ can be deduced from S' by hyper-E-resolution.

Proof. The proof is again by induction on $k(S)$.

(i) If $k(S) = 0$ then S' consists entirely of unit clauses. By theorem 1, we know there is a deduction of $\square$ from S' by E-resolution. But for unit clauses it can easily be verified that any proof by E-resolution is in fact a proof by hyper-E-resolution. For example, one needs to verify that all paramodulations occurring in the proof occur only from clauses consisting entirely of positive equality literals. But for unit clauses this is trivial since any paramodulation which occurs must occur from a clause consisting of entirely (one) positive equality literals.

(ii) The induction step $(k(S) = N)$ follows the general outline of theorem 2 of Reference 1 with the word "hyper-resolution". changed to "hyper-E-resolution." But it is necessary to use additional care in the selection of the clasue C to be split and the literal l to be split out of C. If S contains a non-unit clause containing a positive equality literal then select that clause as C and split the positive equality literal out of C. Then since the literal split out of C is a positive equality literal adding it back will not affect the fact that all previous hyper-E-resolvents are still hyper-E-resolvents. This is so since adding a positive equality literal back will leave all clauses in the deduction with the same negative literals as before (which is crucial for the definition) and because any paramodulation which occurred from clauses consisting entirely of positive equality literals will still occur from clauses consisting of entirely positive equality literals (since the literal added back is a positive equality literal). Thus in this case the induction step can be carried through.

If all positive equality literals of S occur in unit clauses and S contains a non-unit clause containing a positive literal then select that clause as the clause to be split and the positive literal as the literal to be split out. Now when that literal is added back all hyper-E-resolvents will remain hyper-E-resolvents (because the literal added back is positive and because it can't affect the paramodulation since all positive equality literals from which paramodulation could occur are in unit clause). And so in this case also the induction step can be carried through.

If all positive literals (both equality and non-equality) occur in unit clauses, then it can be verified that $\square$ is an immediate hyper-E-resolvent of some clauses in S. This is the case for the following reason. We know that E-resolution is complete and thus there is a deduction of $\square$ from S by E-resolution. But when all positive literals occur in unit clauses it is easy to see that any deduction of $\square$ by E-resolution can be converted into a single step hyper-E-resolution deduction of $\square$.

LIFTING THE PRECEDING RESULTS TO THE GENERAL LEVEL

The completeness results of the preceding section were all established for ground clauses and need to be "lifted" to the general level. Robinson and Wos[5] discuss the problem of lifting for paramodulation and show by an example that the lifting lemma needed does not hold (in contrast to resolution alone where the lifting is straightforward.[7] In Reference 11 they show in great detail that if functional reflexivity units (unit clauses $(x = x)$ and $(f(x_1, \cdots, x_n) = f(x_1, \cdots, x_n))$ for each function symbol are present then the ground completeness results for paramodulation and resolution do lift to the general level. Since E-resolution has been defined in this paper in terms of paramodulation and resolution their proof can easily be carried over to give that the ground completeness results of theorems 1–3 all lift to the general level for functionally reflexive systems. Whether or not paramodulation or E-resolution are complete without functional reflexivity is not known. However, the following simple example does show that neither paramodulation with set of support nor E-resolution with set of support is complete with-

out functional reflexivity. Let

$$S = \{(a = b), (\sim Qg(b)g(g(a))), (x = x), (Q\,x\,g(x))\}$$

and

$$T = \{(Q\,x\,g(x))\}.$$

S is R-unsatisfiable and $S - T$ is a R-satisfiable. However $\square$ cannot be deduced by either paramodulation with T as set of support or by E-resolution with T as set of support. Of course if we add the functional reflexivity unit, $(g(x) = g(x))$, to S then both paramodulation and E-resolution can be used to deduce $\square$ with T as set of support. The completeness of hyper-E-resolution without functional reflexivity is unknown.

FURTHER NOTES ON E-RESOLUTION

The only purpose which the clause $(x = x)$ serves in deductions using E-resolution (or paramodulation) is to resolve against, and thus eliminate trivial inequalities of the form $t \neq t$. It is in fact more efficient to simply put in a special rule for eliminating trivial inequalities rather than always adding the clause $(x = x)$ to the set of R-unsatisfiable clauses, and that is what was done in the original definition and implementation of E-resolution discussed in Reference 4. One can also increase the efficiency of E-resolution by always trying to unify the two literals upon which one is tryir g to E-resolve by means of the unification algorithm first and only use paramodulation to try and unify those terms in the two selected literals which are not unified by this initial attempt at unification. This is what was done in Reference 4. Unfortunately such a system is incomplete (though it may in fact be the better system to use) as shown by the following example.

Let $S = \{(Pf(x)g(y) \vee Ph(x)i(y)), (\sim Pf(a)g(b)), (\sim Ph(b)i(b)), (f(a) = f(c)), (h(c) = h(b))\}$

This set S is R-unsatisfiable but $\square$ cannot be deduced by the technique given in Reference 4, for the only E-resolvents obtainable by the technique (where one always tries straight unification before paramodulating) $(Ph(a)i(b))$ and $(Pf(b)g(b))$. Of course with the modified definition of E-resolution given in this paper one can get a deduction of $\square$ by E-resolution.

REFERENCES

1 R ANDERSON W W BLEDSOE
A linear format for resolution with merging and a new technique for establishing completeness
To appear in Journal of the ACM

2 J L DARLINGTON
Automatic theorem-proving with equality substitutions and mathematical induction
Machine Intelligence Vol 3 B D Michie ed

3 R KOWALSKI
The case for using equality axioms in automatic demonstration
A Paper presented at the Symposium on Automatic Demonstration Paris December 16–21 1968

4 J B MORRIS
E-resolution: Extensions of resolution to include the equality relation
Proc International Joint Conference on Artificial Intelligence Washington D C May 7–9 1969

5 G A ROBINSON L WOS
Paramodulation and theorem-proving in first-order theories with equality
Machine Intelligence Vol 4 B Meltzer and D Michie eds

6 G A ROBINSON L WOS
Completeness of paramodulation
Spring 1968 Meeting of Assn for Symbolic Logic—Abstract to appear in Journal Symbolic Logic

7 J A ROBINSON
A machine oriented logic based on the resolution principle
Journal of the ACM Vol 12 No 1 pp 23–41 January 1965

8 J A ROBINSON
Automatic deduction with hyper-resolution
Int J Computer Math Vol 1 pp 227-234 1965

9 J A ROBINSON
The generalized resolution principle
Machine Intelligence Vol 3 D Michie ed

10 E E SIBERT
A machine-oriented logic incorporating the equality relation
Machine Intelligence Vol 4 B Meltzer and D Michie eds

11 L WOS G A ROBINSON
Paramodulation and set of support
A Paper presented at the IRIA Symposium on Automatic Demonstration at Versailles France December 16–21 1968

A new architecture for mini-computers— The DEC PDP-11

by G. BELL,* R. CADY, H. McFARLAND, B. DELAGI, J. O'LAUGHLIN and R. NOONAN

Digital Equipment Corporation
Maynard, Massachusetts

and

W. WULF

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

The mini-computer** has a wide variety of uses: communications controller; instrument controller; large-system pre-processor; real-time data acquisition systems . . .; desk calculator. Historically, Digital Equipment Corporation's PDP-8 Family, with 6,000 installations has been the archetype of these mini-computers.

In some applications current mini-computers have limitations. These limitations show up when the scope of their initial task is increased (e.g., using a higher level language, or processing more variables). Increasing the scope of the task generally requires the use of more comprehensive executives and system control programs, hence larger memories and more processing. This larger system tends to be at the limit of current mini-computer capability, thus the user receives diminishing returns with respect to memory, speed efficiency and program development time. This limitation is not surprising since the basic architectural concepts for current mini-computers were formed in the early 1960's. First, the design was constrained by cost, resulting in rather simple processor logic and register configurations. Second, application experience was not available. For example, the early constraints often created computing designs with what we now consider weaknesses:

1. limited addressing capability, particularly of larger core sizes
2. few registers, general registers, accumulators, index registers, base registers
3. no hardware stack facilities
4. limited priority interrupt structures, and thus slow context switching among multiple programs (tasks)
5. no byte string handling
6. no read only memory facilities
7. very elementary I/O processing

---

* Also at Carnegie-Mellon University, Pittsburgh, Pennsylvania.

** The PDP-11 design is predicated on being a member of one (or more) of the micro, midi, mini, . . ., maxi (computer name) markets. We will define these names as belonging to computers of the third generation (integrated circuit to medium scale integrated circuit technology), having a core memory with cycle time of .5 ~ 2 microseconds, a clock rate of 5 ~ 10 *Mhz* . . ., a single processor with interrupts and usually applied to doing a particular task (e.g., controlling a memory or communications lines, pre-processing for a larger system, process control). The specialized names are defined as follows:

	maximum addressable primary memory (words)	*processor and memory cost (1970 kilodollars)*	*word length (bits)*	*processor state (words)*	*data types*
micro	8 K	~ 5	8 ~ 12	2	integers, words, boolean vectors
mini	32 K	5 ~ 10	12 ~ 16	2–4	vectors (i.e., indexing)
midi	65 ~ 128 K	10 ~ 20	16 ~ 24	4–16	double length floating point (occasionally)

8. no larger model computer, once a user outgrows a particular model
9. high programming costs because users program in machine language.

In developing a new computer the architecture should at least solve the above problems. Fortunately, in the late 1960's integrated circuit semiconductor technology became available so that newer computers could be designed which solve these problems at low cost. Also, by 1970 application experience was available to influence the design. The new architecture should thus lower programming cost while maintaining the low hardware cost of mini-computers.

The DEC PDP-11, Model 20 is the first computer of a computer family designed to span a range of functions and performance. The Model 20 is specifically discussed, although design guidelines are presented for other members of the family. The Model 20 would nominally be classified as a third generation (integrated circuits), 16-bit word, 1 central processor with eight 16-bit general registers, using two's complement arithmetic and addressing up to 2^{16} eight bit bytes of primary memory (core). Though classified as a general register processor, the operand accessing mechanism allows it to perform equally well as a 0-(stack), 1-(general register) and 2-(memory-to-memory) address computer. The computer's components (processor, memories, controls, terminals) are connected via a single switch, called the Unibus.

The machine is described using the PMS and ISP notation of Bell and Newell (1970) at different levels. The following descriptive sections correspond to the levels: external design constraints level; the PMS level—the way components are interconnected and allow information to flow; the program level or ISP (Instruction Set Processor)—the abstract machine which interprets programs; and finally, the logical design level. (We omit a discussion of the circuit level—the PDP-11 being constructed from TTL integrated circuits.)

DESIGN CONSTRAINTS

The principal design objective is yet to be tested; namely, do users like the machine? This will be tested both in the market place and by the features that are emulated in newer machines; it will indirectly be tested by the life span of the PDP-11 and any offspring.

Word length

The most critical constraint, word length (defined by IBM) was chosen to be a multiple of 8 bits. The memory word length for the Model 20 is 16 bits, although there are 32- and 48-bit instructions and 8- and 16-bit data. Other members of the family might have up to 80 bit instructions with 8-, 16-, 32-and 48-bit data. The internal, and preferred external character set was chosen to be 8-bit ASCII.

Range and performance

Performance and function range (extendability) were the main design constraints; in fact, they were the main reasons to build a new computer. DEC already has (4) computer families that span a range* but are incompatible. In addition to the range, the initial machine was constrained to fall within the small-computer product line, which means to have about the same performance as a PDP-8. The initial machine outperforms the PDP-5, LINC, and PDP-4 based families. Performance, of course, is both a function of the instruction set and the technology. Here, we're fundamentally only concerned with the instruction set performance because faster hardware will always increase performance for any family. Unlike the earlier DEC families, the PDP-11 had to be designed so that new models with significantly more performance can be added to the family.

A rather obvious goal is maximum performance for a given model. Designs were programmed using benchmarks, and the results compared with both DEC and potentially competitive machines. Although the selling price was constrained to lie in the $5,000 to $10,000 range, it was realized that the decreasing cost of logic would allow a more complex organization than earlier DEC computers. A design which could take advantage of medium- and eventually large-scale integration was an important consideration. First, it could make the computer perform well; and second, it would extend the computer family's life. For these reasons, a general registers organization was chosen.

Interrupt response

Since the PDP-11 will be used for real time control applications, it is important that devices can communicate with one another quickly (i.e., the response time of a request should be short). A multiple priority level, nested interrupt mechanism was selected; additional priority levels are provided by the physical position of a device on the Unibus. Software polling is

* PDP-4, 7, 9, 15 family; PDP-5, 8, 8/S, 8/I, 8/L family; LINC, PDP-8/LINC, PDP-12 family; and PDP-6, 10 family. The initial PDP-1 did not achieve family status.

unnecessary because each device interrupt corresponds to a unique address.

Software

The total system including software is of course the main objective of the design. Two techniques were used to aid programmability: first benchmarks gave a continuous indication as to how well the machine interpreted programs; second, systems programmer continually evaluated the design. Their evaluation considered: what code the compiler would produce; how would the loader work; ease of program relocability; the use of a debugging program; how the compiler, assembler and editor would be coded—in effect, other benchmarks; how real time monitors would be written to use the various facilities and present a clean interface to the users; finally the ease of coding a program.

Modularity

Structural flexibility (sometimes called modularity) for a particular model was desired. A flexible and straightforward method for interconnecting components had to be used because of varying user needs (among user classes and over time). Users should have the ability to configure an optimum system based on cost, performance and reliability, both by interconnection and, when necessary, constructing new components. Since users build special hardware, a computer should be easily interfaced. As a by-product of modularity, computer components can be produced and stocked, rather than tailor-made on order. The physical structure is almost identical to the PMS structure discussed in the following section; thus, reasonably large building blocks are available to the user.

Microprogramming

A note on microprogramming is in order because of current interest in the "firmware" concept. We believe microprogramming, as we understand it (Wilkes, 1951), can be a worthwhile technique as it applies to processor design. For example, microprogramming can probably be used in larger computers when floating point data operators are needed. The IBM System/360 has made use of the technique for defining processors that interpret both the System/360 instruction set and earlier family instruction sets (e.g., 1401, 1620, 7090). In the PDP-11 the basic instruction set is quite straightforward and does not necessitate microprogrammed interpretation. The processor-memory connection is asynchronous and therefore memory of any speed can be connected. The instruction set encourages the user to write reentrant programs; thus, read-only memory can be used as part of primary memory to gain the permanency and performance normally attributed to microprogramming. In fact, the Model 10 computer which will not be further discussed has a 1024-word read only memory, and a 128-word read-write memory.

Understandability

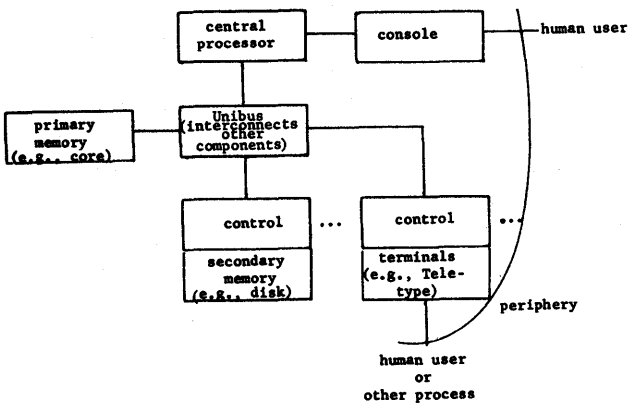
Understandability was perhaps the most fundamental constraint (or goal) although it is now somewhat less important to have a machine that can be quickly understood by a novice computer user than it was a few years ago. DEC's early success has been predicated on selling to an intelligent but inexperienced user. Understandability, though hard to measure, is an important goal because all (potential) users must understand the computer. A straightforward design should simplify the systems programming task; in the case of a compiler, it should make translation (particularly code generation) easier.

PDP-11 STRUCTURE AT THE PMS LEVEL*

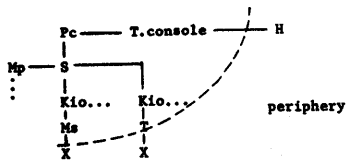
Introduction

PDP-11 has the same organizational structure as nearly all present day computers (Figure 1). The primitive PMS components are: the primary memory (Mp) which holds the programs while the central processor (Pc) interprets them; io controls (Kio) which manage data transfers between terminals (T) or secondary memories (Ms) to primary memory (Mp); the components outside the computer at periphery (X) either humans (H) or some external process (e.g., another computer); the processor console (T. console) by which humans communicate with the computer and observe its behavior and affect changes in its state; and a switch (S) with its control (K) which allows all the other components to communicate with one another. In the case of PDP-11, the central logical switch structure is implemented using a bus or chained switch (S) called the Unibus, as shown in Figure 2. Each physical component has a switch for placing messages on the bus or taking messages off the bus. The central control decides the next component to

* A descriptive (block-diagram) level (Bell and Newell, 1970) to describe the relationship of the computer components: processors memories, switches, controls, links, terminals and data operators.



Conventional block diagram



PMS diagram[1]

[1] PMS Notation

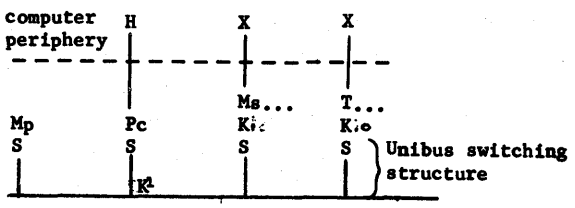
form	comment
Component/X	name X is an alias (abbreviation for a component is separated by /)
a := b	a is assigned the meaning of b
|	delimits mutually exclusive alternatives
Components := (Processor/P | Memory/M|Switch/S| Control/K|Terminal/T| Data operation/D|Link/L| Human/H)	set of primitive components and their abbreviations
$X(a_1:v_1; a_2:v_2; \ldots a_n:v_b)$	n attribute/a, value/v pairs. Attribute may be omitted if it can be inferred from dimensions of value.
index number/#	attribute giving component number
name/'	attribute giving component name
miscellaneous abbreviations := (Mp/primary memory|Ms/secondary memory|Pc/central processor| Kio/io control|Pio/io processor|s/sec/seconds|char/character| b/bit|w/word|i/information|	
———	information carrying link (bi-directional)
→←	uni-directional information carrying links
	delimits alternatives

Figure 1—Conventional block diagram and PMS diagram of PDP-11

use the bus for a message (call). The S (Unibus)differs from most switches because any component can communicate with any other component.

The types of messages in the PDP-11 are along the lines of the hierarchical structure common to present day computers. The single bus makes conventional and other structures possible. The message processes in the structure which utilize S(Unibus) are:

1. The central processor (Pc) requests that data be read or written from or to primary memory (Mp) for instructions and data. The processor calls a particular memory module by concurrently specifying the module's address, and the address within the modules. Depending on whether the processor requests reading or writing, data is transmitted either from the memory to the processor or vice versa.
2. The central processor (Pc) controls the initialization of secondary memory (Ms) and terminal (T) activity. The processor sets status bits in the control associated with a particular Ms or T, and the device proceeds with the specified action (e.g., reading a card, or punching a character into paper tape). Since some devices transfer data vectors directly to primary memory, the vector control information (i.e., the memory location and length) is given as initialization information.
3. Controls request the processor's attention in the form of interrupts. An interrupt request to the processor has the effect of changing the state of the processor; thus the processor begins executing a program associated with the interrupting process. Note, the interrupt process is only a signaling method, and when the processor interruption occurs, the interruptee specifies a unique address value to the processor. The address is a starting address for a program.
4. The central processor can control the transmission of data between a control (for T or Ms) and either the processor or a primary memory for program controlled data transfers. The device signals for attention using the interrupt dialogue and the central processor responds by managing the data transmission in a fashion similar to transmitting initialization information.



[1] Unibus control packaged with Pc

Figure 2—PDP-11 physical structure PMS diagram

5. Some device controls (for T or Ms) transfer data directly to/from primary memory without central processor intervention. In this mode the device behaves similar to a processor; a memory address is specified, and the data is transmitted between the device and primary memory.
6. The transfer of data between two controls, e.g., a secondary memory (disk) and say a terminal/T. display is not precluded, provided the two use compatible message formats.

As we show more detail in the structure there are, of course, more messages (and more simultaneous activity). The above does not describe the shared control and its associated switching which is typical of a magnetic tape and magnetic disk secondary memory systems. A control for a DECtape memory (Figure 3) has an S('DECtape bus) for transmitting data between

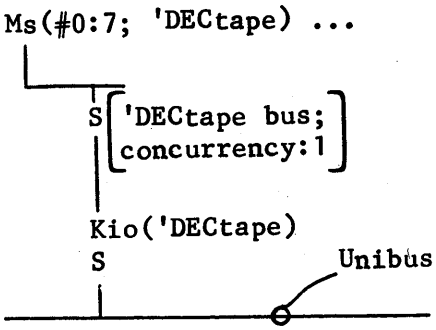


Figure 3—DECtape control switching PMS diagram

a single tape unit and the DECtape transport. The existence of this kind of structure is based on the relatively high cost of the control relative to the cost of the tape and the value of being able to run concurrently with other tapes. There is also a dialogue at the periphery between X-T and X-Ms which does not use the Unibus. (For example, the removal of a magnetic tape reel from a tape unit or a human user (H) striking a typewriter key are typical dialogues.)

All of these dialogues lead to the hierarchy of present computers (Fig. 4). In this hierarchy we can see the paths by which the above messages are passed (Pc-Mp; Pc-K; K-Pc; Kio-T and Kio-Ms; and Kio-Mp; and, at the periphery, T-X and T-Ms; and T.console-H).

Model 20 implementation

Figure 5 shows the detailed structure of a uniprocessor, Model 20 PDP-11 with its various

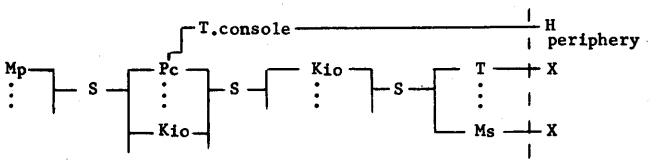
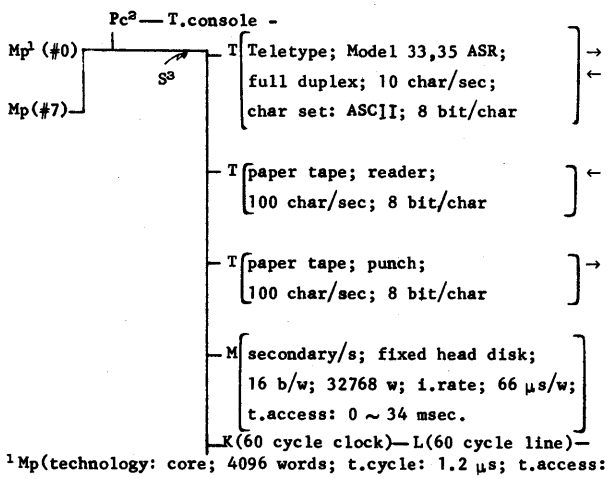


Figure 4—Conventional hierarchy computer structure

components (options). In Figure 5 the Unibus characteristics are surpressed. (The detailed properties of the switch are described in the logical design section.)

Extensions to increase performance

The reader should note (Figure 5) that the important limitations of the bus are: a concurrency of one, namely, only one dialogue can occur at a given time, and a maximum transfer rate of one 16-bit word per .75 μsec., giving a transfer rate of 21.3 megabits/second. While the bus is not a limit for a uni-processor structure, it is a limit for multiprocessor structures. The bus also imposes an artificial limit on the system performance when high speed devices (e.g., TV cameras, disks) are



[1] Mp(technology: core; 4096 words; t.cycle: 1.2 μs; t.access: .6 μs; 16 bits/word)

[2] P(central/c; Model 30; integrated circuit; general registers; 2 addresses/instruction; addresses are: register, stack, Mp; data types: bits, bytes, words, word integers, byte integers, boolean vectors; 8 bits/byte; 16 bits/word operations: (+, -, / (optional), x (optional), /2, x2, ¬, - (negate); ∨, ⊃);
M(processor state; 'general registers; 8 + 1 word; integrated circuit))

[3] S('Unibus; non-hierarchy; bus; concurrency:1; 1 word/.75 μs)

Figure 5—PDP-11 structure and characteristics PMS diagram

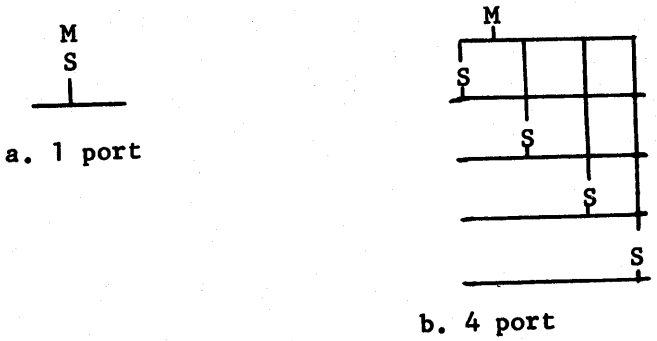


Figure 6—1 and 4 port memory modules PMS diagram

transferring data to multiple primary memories. On a larger system with multiple independent memories the supply of memory cycles is 17 megabits/second times the number of modules. Since there is such a large supply of memory cycles/second and since the central processor can only absorb approximately 16 megabits/second, the simple one Unibus structure must be modified to make the memory cycles available. Two changes are necessary: first, each of the memory modules have to be changed so that multiple units can access each module on an independent basis; and second, there must be independent control accessing mechanisms. Figure 6 shows how a single memory is modified to have more access ports (i.e., connect to 4 Unibusses).

Figure 7 shows a system with 3 independent memory modules which are accessed by 2 independent Unibusses. Note that two of the secondary memories and one of the transducers are connected to both Unibusses. It should be noted that devices which can potentially interfere with Pc-Mp accesses are constructed with two ports; for simple systems, the two ports are both connected to the same bus, but for systems with more busses, the second connection is to an independent bus.

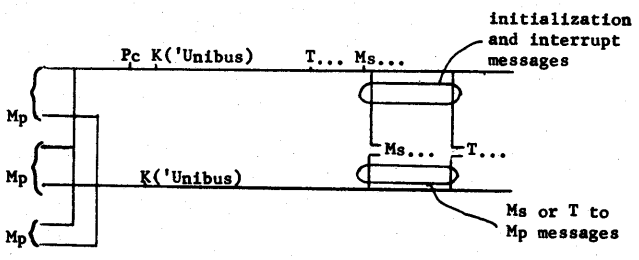


Figure 7—Three Mp, 2 S('Unibus) structure PMS diagram

Figure 8 shows a multiprocessor system with two central processors and three Unibusses. Two of the Unibus controls are included within the two processors, and the third bus is controlled by an independent control unit. The structure also has a second switch to allow either of two processors (Unibusses) to access common shared devices. The interrupt mechanism allows either processor to respond to an interrupt and similarly either processor may issue initialization information on an anonymous basis. A control unit is needed so that two processors can communicate with one another; shared primary memory is normally used to carry the body of the message. A control connected to two Pc's (see Figure 8) can be used for reliability; either processor or Unibus could fail, and the shared Ms would still be accessible.

Higher performance processors

Increasing the bus width has the greatest effect on performance. A single bus limits data transmission to 21.4 megabits/second, and though Model 20 memories are 16 megabits/second, faster (or wider) data path width modules will be limited by the bus. The Model 20 is not restricted, but for higher performance processors operating on double word (fixed point) or triple word (floating point) data two or three accesses are required for a single data type. The direct method to improve the performance is to double or triple the primary memory and central processor data path widths. Thus, the bus data rate is automatically doubled or tripled.

For 32- or 48-bit memories a coupling control unit is needed so that devices of either width appear isomorphic to one another. The coupler maps a data

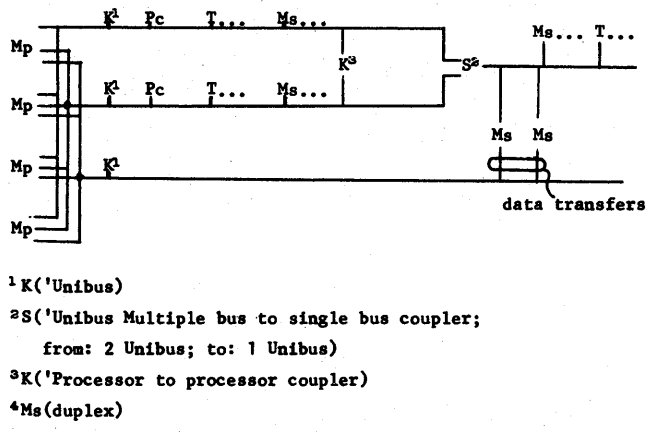


Figure 8—Dual Pc multiprocessor system PMS diagram

request of a given width into a higher- or lower-width request for the bus being coupled to, as shown in Figure 9. (The bus is limited to a fixed number of devices for electrical reasons; thus, to extend the bus a bus repeating unit is needed. The bus repeating control unit is almost identical to the bus coupler.) A computer with a 48-bit primary memory and processor and 16-bit secondary memory and terminals (transducers) is shown in Figure 9.

In summary, the design goal was to have a modular structure providing the final user with freedom and flexibility to match his needs. A secondary goal of the Unibus is open-endedness by providing multiple busses and defining wider path busses. Finally, and most important, the Unibus is straightforward.

THE INSTRUCTION SET PROCESSOR (ISP) LEVEL-ARCHITECTURE*

Introduction, background and design constraints

The Instruction Set Processor (ISP) is the machine defined by hardware and/or software which interprets programs. As such, an ISP is independent of technology and specific implementations.

The instruction set is one of the least understood aspects of computer design; currently it is an art. There is currently no theory of instruction sets, although there have been attempts to construct them (Maurer, 1966), and there has also been an attempt to have a computer program design an instruction set (Haney, 1968). We have used the conventional approach in this design: first a basic ISP was adopted and then incremental design modifications were made (based on the results of the benchmarks).**

Although the approach to the design was conventional, the resulting machine is not. A common classification of processors is as zero-, one-, two-, three-, or three-plus-one-address machines. This scheme has the the form:

$$op\ l1,\ l2,\ l3,\ l4$$

where $l1$ specifies the location (address) in which to store the result of the binary operation (op) of the contents of operand locations $l2$ and $l3$, and $l4$ specifies the location of the next instruction.

The action of the instruction is of the form:

$$l1 \leftarrow l2\ op\ l3;\ goto\ l4$$

The other addressing schemes assume specific values for one or more of these locations. Thus, the one-address von Neumann (Burks, Goldstine and von Neumann, 1946) machines assume $l1 = l2 =$ the "accumulator" and $l4$ is the location following that of the current instruction. The two-address machine assumes $l1 = l2$; $l4$ is the next address.

Historically, the trend in machine design has been to move from a 1 or 2 word accumulator structure as in the von Neumann machine towards a machine with accumulator and index register(s).* As the number of registers is increased the assignment of the registers to specific functions becomes more undesirable and inflexible; thus, the general-register concept has developed. The use of an array of general registers in the processor was apparently first used in the first-generation, vacuum-tube machine, PEGASUS (Elliott et al., 1956) and appears to be an outgrowth of both 1- and 2-address structures. (Two alternative structures—the early 2- and 3-address per instruction computers may be disregarded, since they tend to always access primary memory for results as well as temporary storage and thus are wasteful of time and memory cycles, and require a long instruction.) The stack concept (zero-address) provides the most efficient

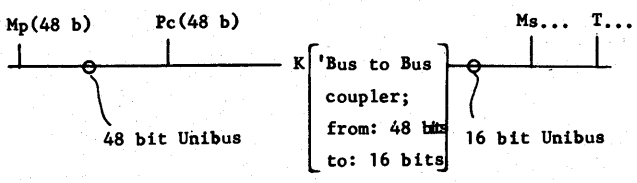


Figure 9—Computer with 48 bit Pc, Mp with 16 bit Ms, T PMS diagram

* The word architecture has been operationally defined (Amdahl, Blaauw and Brooks, 1964) as "the attributes of a system as seen by a programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design and the physical implementation."

** A predecessor multiregister computer was proposed which used a similar design process. Benchmark programs were coded on each of 10 "competitive" machines, and the object of the design was to get a machine which gave the best score on the benchmarks. This approach had several fallacies: the machine had no basic character of its own; the machine was difficult to program since the multiple registers were assigned to specific functions and had inherent idiosyncrasies to score well on the benchmarks; the machine did not perform well for programs other than those used in the benchmark test; and finally, compilers which took addvantage of the machine appeared to be difficult to write. Since all "competitive machines" had been hand-coded from a common flowchart rather than separate flowcharts for each machine, the apparent high performance may have been due to the flowchart organization.

* Due in part to needs, but mainly technology which dictates how large the structure can be.

access method for specifying algorithms, since very little space, only the access addresses and the operators, needs to be given. In this scheme the operands of an operator are always assumed to be on the "top of the stack". The stack has the additional advantage that arithmetic expression evaluation and compiler statement parsing have been developed to use a stack effectively. The disadvantage of the stack is due in part to the nature of current memory technology. That is, stack memories have to be simulated with random access memories, multiple stacks are usually required, and even though small stack memories exist, as the stack overflows, the primary memory (core) has to be used.

Even though the trend has been toward the general register concept (which, of course, is similar to a two address scheme in which one of the addresses is limited to small values), it is important to recognize that any design is a compromise. There are situations for which any of these schemes can be shown to be "best". The IBM System/360 series uses a general register structure, and their designers (Amdahl, Blaauw and Brooks, 1964) claim the following advantages for the scheme:

1. Registers can be assigned to various functions: base addressing, address calculation, fixed point arithmetic and indexing.
2. Availability of technology makes the general registers structure attractive.

The System/360 designers also claim that a stack organized machine such as the English Electric KDF 9 (Allmark and Lucking, 1962) or the Burroughs B5000 (Lonegran and King, 1961) has the following disadvantages:

1. Performance is derived from fast registers, not the way they are used.
2. Stack organization is too limiting and requires many copy and swap operations.
3. The overall storage of general registers and stack machines are the same, considering point #2.
4. The stack has a bottom, and when placed in slower memory there is a performance loss.
5. Subroutine transparency is not easily realized with one stack.
6. Variable length data is awkward with a stack.

We generally concur with points 1, 2, and 4. Point 5 is an erroneous conclusion, and point 6 is irrelevant (that is, general register machines have the same problem). The general-register scheme also allows processor implementations with a high degree of parallelism since instructions of a local block all can operate on several registers concurrently. A set of truly general purpose registers should also have additional uses. For example, in the DEC PDP-10, general registers are used for address integers, indexing, floating point, boolean vectors (bits), or program flags and stack pointers. The general registers are also addressable as primary memory, and thus, short program loops can reside within them and be interpreted faster. It was observed in operation that PDP-10 stack operations were very powerful and often used ((accounting for as many as 20% of the executed instructions, in some programs, e.g., the compilers.)

The basic design decision which sets the PDP-11 apart was based on the observation that by using *truly* general registers and by suitable addressing mechanisms it was possible to consider the machine as a zero-address (stack), one-address (general register), or two-address (memory-to-memory) computer. Thus, it is possible to use whichever addressing scheme, or mixture of schemes, is most appropriate.

Another important design decision for the instruction set was to have only a few data types in the basic machine, and to have a rather complete set of operations for each data type. (Alternative designs might have more data types with few operations, or few data types with few operations.) In part, this was dictated by the machine size. The conversion between data types must be easily accomplished either automatically or with 1 or 2 instructions. The data types should also be sufficiently primitive to allow other data types to be defined by software (and by hardware in more powerful versions of the machine). The basic data type of the machine is the 16 bit integer which uses the two's complement convention for sign. This data type is also identical to an address.

PDP-11 model 20 instruction set (basic instruction set)

A formal description of the basic instruction set is given in Appendix 1 using the ISPL notation (Bell and Newell, 1970). The remainder of this section will discuss the machine in a conventional manner.

Primary memory

The primary memory (core) is addressed as either 2^{16} bytes or 2^{15} words using a 16 bit number. The linear address space is also used to access the input-output devices. The device state, data and control registers are read or written like normal memory locations.

General register

The general registers are named: R[0:7]⟨15:0⟩*; that is, there are 8 registers each with 16 bits. The naming is done starting (at the left with bit 15 (the sign bit) to the least significant bit 0. There are synonyms for R[6] and R[7]:

Stack Pointer/SP⟨15:0⟩ := R[6]⟨15:0⟩
: used to access a special stack which is used to store the state of interrupts, traps and subroutine calls

Program Counter/PC⟨15:0⟩ := R[7]⟨15:0⟩
: points to the current instruction being interpreted. It will be seen that the fact that PC is one of the general registers is crucial to the design.

Any general register, R[0:7], can be used as a stack pointer. The special Stack Pointer (SP) has additional properties that force it to be used for changing processor state interrupts, traps, and subroutine calls (It also can be used to control dynamic temporary storage subroutines.)

In addition to the above registers there are 8 bits used (from a possible 16) for processor status, called PS⟨15.0⟩ register. Four bits are the Condition Codes (CC) associated with arithmetic results; the T-bit controls tracing; and three bits control the priority of running programs Priority ⟨2:0⟩. Individual bits are mapped in PS as shown in Appendix 1.

Data types and primitive operations

There are two data lengths in the basic machine: bytes and words, which are 8 and 16 bits, respectively. The non-trivial data types are word length integers (w.i.); byte length integers (by .i); word length boolean vectors (w.bv), i.e., 16 independent bits (booleans) in a 1 dimensional array; and byte length boolean vectors (by.bv). The operations on byte and word boolean vectors are identical. Since a common use of a byte is to hold several flag bits (booleans), the operations can be combined to form the complete set of 16 operations. The logical operations are: "clear," "complement," "inclusive or," and "implication" ($x \supset y$ or $\neg x \vee y$).

There is a complete set of arithmetic operations for the word integers in the basic instruction set. The arithmetic operations are: add, subtract, multiply (optional), divide (optional), compare, add one, subtract one, clear, negate, and multiply and divide by powers of two (shift). Since the address integer size is 16 bits, these data types are most important. Byte length integers are operated on as words by moving them to the general registers where they take on the value of word integers. Word length integer operations are carried out and the results are returned to memory (truncated).

The floating point instructions defined by software (not part of the basic instruction set) require the definition of two additional data types (of length two and three), i.e., double word (d.w.) and triple (t.w.) words. Two additional data types, double integer (d.i.) and triple floating point (t.f. or f) are provided for arithmetic. These data types imply certain additional operations and the conversion to the more primitive data types.

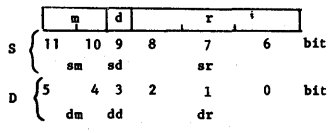
Address (operand) calculation

The general methods provided for accessing operands are the most interesting (perhaps unique) part of the machine's structure. By defining several access methods to a set of general registers, to memory, or to a stack (controlled by a general register), the computer is able to be a 0, 1 and 2 address machine. The encoding of the instruction Source (S) fields and Destination (D) fields are given in Fig. 10 together with a list of the various access modes that are possible. (Appendix 1 gives a formal description of the effective address calculation process.)

It should be noted from Figure 10 that all the common access modes are included (direct, indirect, immediate, relative, indexed, and indexed indirect) plus several relatively uncommon ones. Relative (to PC) access is used to simplify program loading, while immediate mode speeds up execution. The relatively uncommon access modes, auto-increment and auto-decrement, are used for two purposes: access to a stack under control of the registers* and access to bytes or words organized as strings or vectors. The indirect access mode allows a stack to hold addresses of data (instead of data). This mode is desirable when manipulating longer and variable-length data types (e.g., strings, double fixed and triple floating point). The register auto increment mode may be used to access a byte string; thus, for example, after each access, the register can be made to point to the next data item. This is used for moving data blocks, searching for particular elements of a vector, and byte-string operations (e.g., movement, comparisons, editing).

* A definition of the ISP notation used here may be found in Appendix 1.

*Note, by convention a stack builds toward register 0, and when the stack crosses 400_8, a stack overflow occurs.



r = register specification R[r]
d = defer (indirect) address bit
m = mode (00 = R[r]; 01 = R[r]; next R[r] +@i;[1]
10 = R[r], R[r] -ai, next R[2]
11 = indexed with next word)

The following access modes can be specified:

0 direct-to a register, R[r]
1 indirect-to a register, R[r] for address of data
2 auto increment via register (pop) - use register as address, then increment register
3 auto increment via register (pop) - defer
4 auto decrement via register (push) - decrement register, then use register as address
5 auto decrement indirect - decrement register, then use register as the address of the address of data

2 immediate data - next full word is the data (r=PC)
3 direct data - next full word is the address of data (r=PC)

6 direct indexed - use next full word indexed with R[r] as address of data
7 direct indexed - indirect - use next full word indexed with R[r] as the address of the address of data
6 relative access - next full word plus PC is the address (r=PC)
7 relative indirect access - next full word plus PC is the address of the address of data (r=PC)

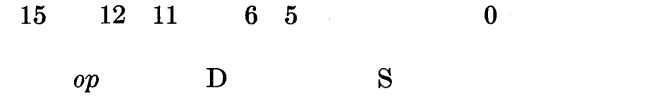
[1]address increment/ai value is 1 or 2

Figure 10—Address calculation formats

This addressing structure provides flexibility while retaining the same, or better, coding efficiency than classical machines. As an example of the flexibility possible, consider the variations possible with the most trivial word instruction MOVE (see Figure 11). The MOVE instruction is coded as it would appear in conventional 2-address, 1-address (general register) and 0-address (stack) computers. The two-address format is particularly nice for MOVE, because it provides an efficient encoding for the common operation: A ← B (note, the stack and general registers are not involved). The vector move A[I] ← B(I) is also efficiently encoded. For the general register (and 1-address format), there are about 13 MOVE operations that are commonly used. Six moves can be encoded for the stack (about the same number found in stack machines).

Instruction formats

There are several instruction decoding formats depending on whether 0, 1, or 2 operands have to be explicitly referenced. When 2 operands are required, they are identified as Source/S and Destination/D and the result is placed at Destination/D. For single operand instructions (unary operators) the instruction action is D ← u D; and for two operand instructions (binary operators) the action is D ← D b S (where u and b are unary and binary operators, e.g., ¬, − and +, −, ×, /, respectively. Instructions are specified by a 16-bit word. The most common binary operator format (that for operations requiring two addresses) is shown below.



The other instruction formats are given in Figure 12.

Instruction interpretation process

The instruction interpretation process is given in Figure 13, and follows the common fetch-execute cycle. There are three major states: (1) interrupting—the PC and PS are placed on the stack accessed by the Stack Pointer/SP, and the new state is taken from an address specified by the source requesting the trap or interrupt; (2) trace (controlled by T-bit)—essentially one instruction at a time is executed as a trace

Assembler Format	Effect	Description
Two Address Machine format:		
MOVE B,A[1]	A ← B	replace A with contents of B
MOVE #N,A	A ← N	replace A with number, N
MOVE B(RZ), A(RZ)	A[I] ← B[I]	replace element of a connector
MOVE (R_3) +, (R_4) +	A[I] ← B[I]; I ← I + 1	replace element of a vector, move to next element
General Register Machine format:		
MOVE A,R1	R1 ← A	load register
MOVE R1, A	A ← R1	store register
MOVE @A,R1	R1 ← M[A]	load or store indirect via element A
MOVE R1, R3	R1 ← R3	register to register transfer
MOVE R1, A(RZ)	A[I] ← R1	store indexed (load indexed) (or store)
MOVE @A(R0),R1	R1 ← M[A[I]]	load (or store) indexed indirect
MOVE (R1), R3	R1 ← M[R2]	load indirect via register
MOVE (R1) +, R3	R3 ← M[I]	load (or store) element indirect via register, move to next element
Stack Machine format:		
MOVE #N, -(R0)	S ← N	load stack with literal
MOVE A, -(R0)	S ← A	load stack with contents of A
MOVE @(R0)+, -(R0)	S ← M[S]	load stack with memory specified by top of stack
MOVE (R0)+, A	A ← S	store stack in A
MOVE (R0)+, @(R0)+	M[S_2] ← S_1	store stack top in memory addressed by stack top -1
MOVE (R0), -(R0)	S ← S	duplicate top of stack

[1]Assembler format:
() denotes contents of memory addressed by
- decrement register first
+ increment register after
@ indirect
# literal

Figure 11—Coding for the MOVE instruction to compare with conventional machines

Binary arithmetic and logical operations: [bop | S | D] [1]

form: D ← S b D

example: ADD (:=bop=0010) → (CC,D ← D+S);

Unary arithmetic and logical operation: [uop | D]

form: D ← u D;

examples: NEG (:=uop=0000101100) → (CC,D ← - D) - negate

ASL (:=uop=0000110011) → (CC,D ← D x 2); shift left

Branch (relative) operators: [brop | offset]

form: if brop condition then (PC ← PC + offset);

example: BEQ (: = brop = 03_{16}) (Z → (PC ← PC + offset)):

Jump: [0 000 000 001 | D]

form: PC ← D + Pc

Jump to subroutine: [0 000 100 | D]

save R[sr] on stack, enter subroutine at D + PC

Misc. operations: [op code]

form: ST ← f

example: HALT (: = instruction = 0) → (RUN ← 0);

[1] Note: these instructions are all 1 word. D and/or S may each require 1 additional immediate data or address word. Thus instructions can be 1, 2, or 3 words long.

Figure 12—PDP-11 instruction formats (simplified)

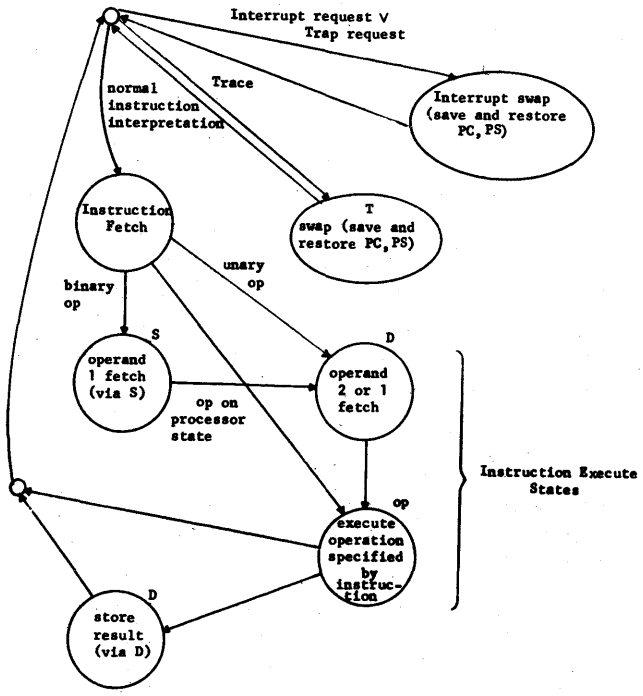


Figure 13—PDP-11 instruction interpretation process state diagram

trap occurs after each instruction, and (3) normal instruction interpretation. The five (lower) states in the diagram are concerned with instruction fetching, operand fetching, executing the operation specified by the instruction and storing the result. The non-trivial details for fetching and storing the operands are not shown in the diagram but can be constructed from the effective address calculation process (Appendix 1). The state diagram, though simplified, is similar to 2- and 3-address computers, but is distinctly different than a 1 address (1 accumulator) computer.

The ISP description (Appendix 1) gives the operation of each of the instructions, and the more conventional diagram (Fig. 12) shows the decoding of instruction classes. The ISP description is somewhat incomplete; for example, the add instruction is defined as: ADD (:= bop = 0010) → (CC,D ← D + S); *addition* does not exactly describe the changes to the Condition Codes/CC (which means whenever a binary opcode [bop] of 0010_2 occurs the ADD instruction is executed with the above effect). In general, the CC are based on the result, that is, Z is set if the result is zero, N if negative, C if a carry occurs, and V if an overflow was detected as a result of the operation. Conditional branch instructions may thus follow the arithmetic instruction to test the results of the CC bits.

Examples of addressing schemes

Use as a stack (zero address) machine

Figure 14 lists typical zero-address machine instructions together with the PDP-11 instructions which perform the same function. It should be noted that translation (compilation) from normal infix expressions to reverse Polish is a comparatively trivial task. Thus, one of the primary reasons for using stacks is for the evaluation of expressions in reverse Polish form.

Consider an assignment statement of the form

$$D \leftarrow A + B/C$$

which has the reverse Polish form

$$DABC/+ \leftarrow$$

and would normally be encoded on a stack machine as follows

load stack address of D
load stack A
load stack B
load stack C
/
+
store

Common stack instruction:	Equivalent PDP-11 instruction:
place address value A on stack	MOVE #A, -(R0)
load stack from memory address specified by stack	MOVE @(R0)+, - (R0)
load stack from memory location A	MOVE A, -(R0)
store stack at memory address specified by stack	MOVE (R0)+, @(R0)+
store stack at memory location A	MOVE (R0)+, A
duplicate top of stack	MOVE (R0), -(R0)
+, add 2 top data of stack to stack	ADD (R0) +, @R0
-, x, /; subtract, multiply, divide	(see add)
-; negate top data of stack	NEG @R0
clear top data of stack	CLR @R0
∨; "inclusive or" 2 top data of stack "and" 2 top data of stack	BSET (R0)+, @R0
¬; complement top of stack	COM @R0
test top of stack (set branch indicators)	TST @R0
branch on indicator	BR (=, ≠, >, ≥, <, ≤)
jump unconditional	JUMP

add addressed location A to top of stack - (not common for stack machine) equivalent to: load stack, add swap top 2 stack data	ADD A, @R0
	MOVE (R0)+, R1
	MOVE (R0)+, R2
	MOVE R1, -(R0)
	MOVE R2, -(R0)

reset stack location to N	MOVE #N, R0 COM @R0
∧, "and" 2 top stack data	BCLR (R0)+, @R0

[1] Stack pointer has been arbitrarily used as register R0 for this example.

Figure 14—Stack computer instructions and equivalent PDP-11 instructions

However, with the PDP-11 there is an address method for improving the program encoding and run time, while not losing the stack concept. An encoding improvement is made by doing an operation to the top of the stack from a direct memory location (while loading). Thus the previous example could be coded as:

load stack B
divide stack by C
add A to stack
store stack D

Use as a one-address (general register) machine

The PDP-11 is a general register computer and should be judged on that basis. Benchmarks have been coded to compare the PDP-11 with the larger DEC PDP-10. A 16 bit processor performs better than the DEC PDP-10 in terms of bit efficiency, but not with time or memory cycles. A PDP-11 with a 32 bit wide memory would, however, decrease time by nearly a factor of two, making the times essentially comparable.

Use as a two-address machine

Figure 15 lists typical two-address machine instructions together with the equivalent PDP-11 instructions for performing the same operations. The most useful instruction is probably the MOVE instruction because it does not use the stack or general registers. Unary instructions which operate on and test primary memory are also useful and efficient instructions.

Extensions of the instruction set for real (floating point) arithmetic

The most significant factor that affects performance is whether a machine has operators for manipulating data in a particular format. The inherent generality of a stored program computer allows any computer by subroutine to simulate another—given enough time and memory. The biggest and perhaps only factor that separates a small computer from a large computer is whether floating point data is understood by the computer. For example, a small computer with a cycle time of 1.0 microseconds and 16 bit memory width might have the following characteristics for a floating point add, excluding data accesses:

programmed:	250 microseconds
programmed (but special normalize and differencing of exponent instructions):	75 microseconds
microprogrammed hardware:	25 microseconds
hardwired:	2 microseconds

It should be noted that the ratios between programmed and hardwired interpretation varies by roughly two orders of magnitude. The basic hardwiring scheme and the programmed scheme should allow binary program compatibility, assuming there is an interpretive program for the various operators in the Model 20. For example, consider one scheme which would add eight 48 bit registers which are addressable in the extended instruction set. The eight floating registers, F, would be mapped into eight double length

Two Address Computer	PDP-11
A ← B; transfer B to A	MOVE B,A
A ← A+B; add	ADD B,A
-, x, /	(see add)
A ← -A; negate	NEG A
A ← A ∨ B; inclusive or	BSETB,A
A ← ¬ A; not	COM
Jump unconditioned	JUMP
Test A, and transfer to B	TST A BR (=, ≠, >, ≥, <, ≤) B

Figure 15—Two address computer instructions and equivalent PDP-11 instructions

(32 bit) registers, D. In order to access the various parts of F or D registers, registers F0 and F1 are mapped onto registers R0 to R2 and R3 to R5.

Since the instruction set operation code is almost completely encoded already for byte and word length data, a new encoding scheme is necessary to specify the proposed additional instructions. This scheme adds two instructions: enter floating point mode and execute one floating point instruction. The instructions for floating point and double word data would be:

binary ops	op	floating point/f	and double word/d
bop′ S D	←	FMOVE	DMOVE
	+	FADD	DADD
	−	FSUB	DSUB
	×	FMUL	DMUL
	/	FDIV	DDIV
	compare	FCMP	DCMP
unary ops			
uop′ D	−	FNEG	DNEG

LOGICAL DESIGN OF S(UNIBUS) AND PC

The logical design level is concerned with the physical implementation and the constituent combinatorial and sequential logic elements which form the various computer components (e.g., processors, memories, controls). Physically, these components are separate and connected to the Unibus following the lines of the PMS structure.

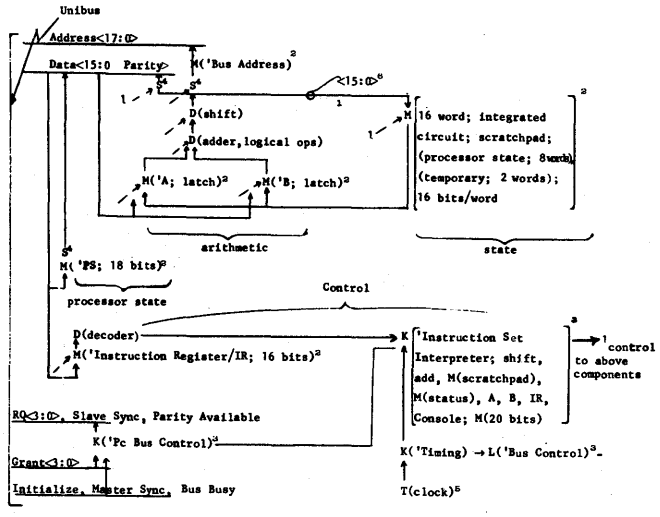
Unibus organization

Figure 16 gives a PMS diagram of the Pc and the entering signals from the Unibus. The control unit for the Unibus, housed in Pc for the Model 20, is not shown in the figure.

The PDP-11 Unibus has 56 bi-directional signals conventionally used for program-controlled data transfers (processor to control), direct-memory data transfers (processor or control to memory) and control-to-processor interrupt. The Unibus is interlocked; thus transactions operate independent of the bus length and response time of the master and slave. Since the bus is bi-directional and is used by all devices, any device can communicate with any other device. The controlling device is the master, and the device to which the master is communicating is the slave. For example, a data transfer from processor (master) to memory (always a slave) uses the Data Out dialogue facility for writing and a transfer from memory to processor uses the Data In dialogue facility for reading.

Bus control

Most of the time the processor is bus master fetching instructions and operands from memory and storing results in memory. Bus mastership is determined by the current processor priority and the priority line upon which a bus request is made and the physical placement of a requesting device on the linked bus.



[1]D - data operations to perform shifting and adding - usually combinatorial
[2]M - memories of a word (16 bits) or several words
[3]K - control units; sequential switching circuits
[4]S - switch, used to gate contents of an M to a set of lines
[5]T - Transducer - to encode time into a logic (clock) signal
[6]Notation to denote 16 lines named 15,14,...,1,0

Figure 16—PDP-11 Pc structure

The assignment of bus mastership is done concurrent with normal communication (dialogues).

Unibus dialogues

Three types of dialogues use the Unibus. All the dialogues have a common protocol which first consists of obtaining the bus mastership (which is done concurrent with a previous transaction) followed by a data exchange with the requested device. The dialogues are: Interrupt; Data In and Date In Pause; and Data Out and Data Out Byte.

Interrupt

Interrupt can be initiated by a master immediately after receiving bus mastership. An address is transmitted from the master to the slave on Interrupt. Normally, subordinate control devices use this method to transmit an interrupt signal to the processor.

Data in and data in pause

These two bus operations transmit slave's data (whose address is specified by the master) to the master. For the Data In Pause operation data is read into the master and the master responds with data which is to be rewritten in the slave.

Data out and data out byte

These two operations transfer data from the master to the slave at the address specified by the master. For Data Out a word at the address specified by the address lines is transferred from master to slave. Data Out Byte allows a single data byte to be transmitted.

Processor logical design

The Pc is designed using TTL logical design components and occupies approximately eight 8″ × 12″ printed circuit boards. The organization of the logic is shown in Figure 17. The Pc is physically connected to two other components, the console and the Unibus. The control for the Unibus is housed in the Pc and occupies one of the printed circuit boards. The most regular part of the Pc, the arithmetic and state section, is shown at the top of the figure. The 16-word scratch-pad memory and combinatorial logic data operators, D(shift) and D(adder, logical ops), form the most regular part of the processor's structure. The 16-word memory holds most of the 8-word processor state found in the ISP, and the 8 bits that form the Status word are stored in an 8-bit register. The input to the adder-shift network has two latches which are either memories or gates. The output of the adder-shift network can be read to either the data or address parts of the Unibus, or back to the scratch-pad array.

The instruction decoding and arithmetic control are less regular than the above data and state and these are shown in the lower part of the figure. There are two major sections: the instruction fetching and decoding control and the instruction set interpreter (which in effect defines the ISP). The later control section operates on, hence controls, the arithmetic and state parts of the Pc. A final control is concerned with the interface to the Unibus (distinct from the Unibus control that is housed in the Pc).

CONCLUSIONS

In this paper we have endeavored to give a complete description of the PDP-11 Model 20 computer at four descriptive levels. These present an unambiguous specification at two levels (the PMS structure and the ISP), and, in addition, specify the constraints for the design at the top level, and give the reader some idea of the implementation at the bottom level logical design. We have also presented guidelines for forming additional models that would belong to the same family.

ACKNOWLEDGMENTS

The authors are grateful to Mr. Nigberg of the technical publication department at DEC and to the reviewers for their helpful criticism. We are especially grateful to Mrs. Dorothy Josephson at Carnegie-Mellon University for typing the notation-laden manuscript.

REFERENCES

1 R H ALLMARK J R LUCKING
Design of an arithmetic unit incorporationg a nesting store
Proc IFIP Congress pp 694–698 1962

2 G M AMDAHL G A BLAAUW F P BROOKS JR
Architecture of the IBM System/360
IBM Journal Research and Development Vol 8 No 2 pp 87–101 April 1964

3 C G BELL A NEWELL
Computer structures
McGraw-Hill Book Company Inc New York In press 1970

4 A W BURKS H H GOLDSTINE J VON NEUMANN
Preliminary discussion of the logical design of an electronic computing instrument, Part II
Datamation Vol 8 No 10 pp 36–41 October 1962
5 W S ELLIOTT C E OWEN C H DEVONALD B G MAUDSLEY
The design philosophy of Pegasus, a quantity-production computer
Proceedings IEEE Pt. B 103 Supp 2 pp 188–196 1956
6 F M HANEY
Using a computer to design computer instruction sets
Thesis for Doctor of Philosophy degree College of Engineering and Science Department of Computer Science Carnegie-Mellon University Pittsburgh Pennsylvania May 1968
7 W LONERGAN P KING
Design of the B5000 system
Datamation Vol 7 No 5 pp 28–32 May 1961
8 W D MAURER
A theory of computer instructions
Journal of the ACM Vol 13 No 2 pp 226–235 April 1966
9 S ROTHMAN
R/W 40 data processing system
International Conference on Information Processing and Auto-math 59 Ramo-Wooldridge (A division of Thompson Ramo Wooldridge Inc) Los Angeles California June 1959
10 M V WILKES
The best way to design an automatic calculating machine
Report of Manchester University Computer Inaugural Conference July 1951 (Manchester 1953)

APPENDIX 1

DEC PDP–11 instruction set processor Description (in ISPL*)

The following description is not a detailed description of the instructions. The description omits the trap behavior of unimplemented instructions, references to non-existent primary memory and io devices, SP (stack) overflow, and power failure.

Primary Memory State

M/Mb/Memory[0:$2^{16}-1$]⟨7:0⟩	*(byte memory)*
Mw[0:$2^{15}-1$]⟨15:0⟩ : = M[0:$2^{16}-1$]⟨7:0⟩	*(word memory mapping)*

Processor State (9 words)

R/Registers[0:7]⟨15:0⟩	*(word general registers)*
SP⟨15:0⟩ : = R[6]⟨15:0⟩	*(stack pointer)*
PC⟨15:0⟩ : = R[7]⟨15:0⟩	*(program counter)*

---

*ISP NOTATION

Although the ISP language has not been described in publications, its syntax is similar to other languages. The language is inherently interpreted in parallel, thus to get sequential evaluation the word "next" must be used. Italics are used for comments. The following notes are in order:

a: = $f(\ldots)$	equivalence or substitution process used for name and process substitution. For every occurrence of a, $f(\ldots)$ replaces it.
$a \leftarrow f(\ldots)$	Replacement operator; the contents in register a are replaced by the value of the function.
register declaration, e.g., $Q[0:1]\,[0:4095]\,\langle 15:0\rangle$	an array of words of two dimensions 2 and 4096; each word has 16 bits denoted 15, 14, 13, . . ., 1, 0
$\langle a:b\rangle_n$	Denotes a range of characters $a, a+1\ldots., b$ to base n. If n is not given, the base is 2.
$[c:d]$	Array designation $c, c+1, \ldots, d$
$a \rightarrow b;$	equivalent to ALGOL if a then b
"next"	sequential interpretation
instruction declaration, e.g., ADD (: = bop = 0010) → $(CC, D \leftarrow D + S)$	defines the "ADD" instruction, assigns it a value, and gives its operation. ADD is executed when bop = 0010_2. Equivalent to: ADD → $(CC, D \leftarrow D + S)$ where ADD: = (bop = 0010) bop has been previously declared
□	concatenation, consider the combined registers as one

operators: = (+/add | −/subtract/negate | ×/multiply | //divide | ∧/and | ∨/or | √/not | ⊕/exclusive or | =/equal/>/greater than | ≥ | < | ≤ | ≠ | modulo | etc.)

PS⟨15:0⟩ (*processor state register*)
- Priority/P⟨2:0⟩ := PS⟨7:5⟩ (*under program control; priority level of the process currently being interpreted a higher level process may interrupt or trap this process*)
- CC/Condition␣Codes⟨3:0⟩ := PS⟨3:0⟩ (*under program control; when set, each instruction executed will trap; used for interpretive and breakpoint debugging*)
  - Carry/C := CC⟨0⟩ (*a result condition code indicating an arithmetic carry from bit 15 of the last operation*)
  - Negative/N := CC⟨3⟩ (*a result condition code indicating last result was negative*)
  - Zero/Z := CC⟨2⟩ (*a result condition code indicating last result was zero*)
  - Overflow/V := CC⟨1⟩ (*a result condition code indicating an arithmetic overflow of the last operation*)
  - Trace/T := ST⟨4⟩ (*denotes whether instruction trace trap is to occur after each instruction is executed*)
  - Undefined⟨7:0⟩ := PS⟨15:8⟩ (*unused*)
- Run (*denotes normal execution*)
- Wait (*denotes waiting for an interrupt*)

Instruction Format
(Bit assignments used in the various instruction formats)
- i/instruction⟨15:0⟩
  - bop⟨3:0⟩ := i⟨15:12⟩ (*binary operation code*)
  - uop⟨15:6⟩ := i⟨15:6⟩ (*unary operation code*)
  - brop⟨15:8⟩ := i⟨15:8⟩ (*branch operation code*)
  - sop⟨15:6⟩ := i⟨15:6⟩ (*shift operation code*)
  - s/source⟨5:0⟩ := i⟨11:6⟩ (*source control byte*)
    - sm⟨0:1⟩ := s⟨5:4⟩ (*source mode control*)
    - sd := s⟨3⟩ (*source defer bit*)
    - sr := s⟨2:0⟩ (*source register*)
  - d/destination⟨5:0⟩ := i⟨5:0⟩ (*destination control byte*)
    - dm⟨0:1⟩ := d⟨5:4⟩
    - dd := d⟨3⟩
    - dr⟨2:0⟩ := d⟨2:0⟩
  - offset⟨7:0 := i⟨7:0⟩ (*signed 7 bit integer*)
- address␣increment/ai (*implicit bit derived from i to denote byte or word length operations*)

Data Types
- by/byte⟨7:0⟩
- w/word⟨15:0⟩
- by.i/byte.integer⟨7:0⟩ (*signed integers*)
- w.i/word.integer⟨15:0⟩
- by.bv/byte.boolean␣vector⟨7:0⟩ (*boolean vectors (bits)*)
- w.bv/word.boolean␣vector⟨15:0⟩

```
d/double␣word⟨31:0⟩                                              (*double word)
t/triple␣word⟨47:0⟩                                              (*triple word)
f/t.f/triple.floating␣point⟨47:0⟩                                (*triple floating point)

Source/S and Destination/D Calculation
S/Source⟨15:0⟩ : = ( ¬ sd → (                                    (direct access)
  (sm = 00) → R[sr];                                             (register)
  (sm = 01) ∧ (sr ≠ 7) → (M[R[sr]]; next R[sr] ← R[sr] + ai);    (auto increment)
  (sm = 01) ∧ (sr = 7) → (M[PC]; PC ← PC + 2);                   (immediate)
  (sm = 10) → (R[sr] ← R[sr] − ai; next M[R[sr]]);               (auto decrement)
  (sm = 11) ∧ (sr ≠ 7) → (M[M[PC] + R[sr]]; PC ← PC + 2);        (indexed)
  (sm = 11) ∧ (sr = 7) → (M[M[PC] + PC]; PC ← PC + 2));          (relative)
                  sd → (                                         (indirect access)
  (sm = 00) → M[R[sr]];                                          (indirect via register)
  (sm = 01) ∧ (sr ≠ 7) → (M[M[R[sr]]]; next R[sr] ← R[sr] + ai); (indirect via stack, auto decrement)
  (sm = 01) ∧ (sr = 7) → (M[M[PC]]; PC ← PC + 2);                (direct absolute)
  (sm = 10) → (R[sr] ← R[sr] − ai; next M[R[sr]]);               (indirect via stack, auto increments)
  (sm = 11) ∧ (sr ≠ 7) → (M[M[PC] + R[sr]]; PC ← PC + 2);        (indirect, indexed)
  (sm = 11) ∧ (sr = 7) → (M[M[M[PC] + PC]]; PC ← PC + 2))        (indirect relative)
```

(The above process defines how operands are determined (accessed) from either memory or the registers. The various length operands, Db(byte), Dw(word), Dd(double) and Df(floating) are not completely defined. The Source/S and Destination/D processes are identical. In the case of jump instruction an address, D′, is used—instead of the word in location M[CI].)

```
Instruction Interpretation Process
¬ Interrupt␣rqs ∧ Run ∧ Wait → (i ← M[PC]; PC ← PC + 2;          (fetch)
                                  next instruction␣execution; next   (execute)
  T → (SP ← SP + 2; next                                         (trace bit store state)
    M[SP] ← PS;
    SP ← SP + 2; next
    M[SP] ← PC;
    PC ← M[14₈]
    ST ← M[16₈]))

  Interrupt␣rq[j] ∧ (CC[j] > CC) ∧ Run → (T ← 0;                 (interrupt)
    SP ← SP + 2; next
    M[SP] ← PS;                                                  (store state and PC enter new process). The locations M[f(j)] are:
                                                                 reserved instruction = M[10]
                                                                 illegal instruction = M[4]
                                                                 stack overflow = M[4]
                                                                 bus errors = M[4])
    SP ← SP + 2;
    M[SP] ← PC
    PC ← M[f(j)]
    PS ← M[f(j) + 2])
Instruction Set and the Execution Process
```

(The following instruction set will be defined briefly and is incomplete. It is intended to give the reader a simple understanding of the machine operation.)

```
Instruction␣execution : = (
  MOV(: = bop = 0001) → (CC,D ← S);                              (move word)
  MOVB(: = bop = 1001) → (CC,Db ← Sb);                           (move byte)
```

* not hardwired or optional

Binary Arithmetic: D ← D b S;

ADD(:= bop = 0110) → (CC,D ← D+s);	(*add*)
SUB(:= bop = 1110) → (CC,D ← D − S);	(*subtract*)
CMP(:= bop = 0010) → (CC ← D − S);	(*word compare*)
CMPB(:= bop = 1010) → (CC ← Db − Sb);	(*byte compare*)
MUL(:= bop = 0111) → (CC,D ← D × S);	(**multiply if D is a register then a double length operator*)
DIV(:= bop = 1111) → (CC,D ← D/S);	(**divide, if D is a register, then a remainder is saved*)

Unary Arithmetic D ← u S;

CLR(:= uop = 050_8) → (CC,D ← 0);	(*clear word*)
CLRB(:= uop = 1050_8) → (CC,Db ← 0);	(*clear byte*)
COM(:= uop = 051_8) → (CC,D ← ¬D);	(*complement word*)
COMB(:= uop = 1051_8) → (CC,Db ← ¬Db);	(*complement byte*)
INC(:= uop = 052_8) → (CC,D ← D + 1);	(*increment word*)
INCB(:= uop = 1052_8) → (CC,Db ← Db + 1);	(*increment byte*)
DEC(:= uop = 053_8) → (CC,D ← D − 1);	(*decrement word*)
DECB(:= uop = 1053_8) → (CC,Db ← Db − 1);	(*decrement byte*)
NEG(:= uop = 054_8) → (CC,D ← −D);	(*negate*)
NEGB(:= uop = 1054_8) → (CC,Db ← −Db)	(*negate byte*)
ADC(:= uop = 055_8) → (CC,D ← D + C);	(*add the carry*)
ADCB(:= uop = 1055_8) → (CC,Db ← Db + C);	(*add to byte the carry*)
SBC(:= uop = 056_8) → (CC,D ← D − C);	(*subtract the carry*)
SBCB(:= uop = 1056_8) → (CC,Db ← Db − C);	(*subtract from byte the carry*)
TST(:= uop = 057_8) → (CC ← D);	(*test*)
TST(:= uop = 1057_8) → (CC ← Db);	(*test byte*)

Shift operations: D ← D × 2^n;

ROR(:= sop = 060_8) → (C□D ← C□D/2{rotate};	(*rotate right*)
RORB(:= sop = 1060_8) → (C□Db ← C□Db/2{rotate});	(*byte rotate right*)
ROL(:= sop = 061_8) → (C□D ← C□D × 2{rotate});	(*rotate left*)
ROLB(:= sop = 1061_8) → (C□Db ← C□Db × 2{rotate});	(*byte rotate left*)
ASR(:= sop = 062_8) → (CC,D ← D × 2);	(*arithmetic shift right*)
ASRB(:= sop = 1062_8) → (CC,Db ← Db/2);	(*byte arithmetic shift right*)
ASL(:= sop = 063_8) → (CC,D ← D × 2);	(*arithmetic shift left*)
ASLB(:= sop = 1063_8) → (CC,Db ← Db × 2);	(*byte arithmetic shift left*)
ROT(:= sop = 064_8) → (C□D ← D × 2^s);	(*rotate*)
ROTB(:= sop = 1064_8) → (C□Db ← D × 2^s);	(*byte rotate*)
LSH(:= sop = 065_8) → (CC,D ← D × 2^s{logical});	(**logical shift*)
LSHB(:= sop = 1065_8) → (CC,Db ← Db × 2^s{logical});	(**byte logical shift*)
ASH(:= sop = 066_8) → (CC,D ← D × 2^s);	(**arithmetic shift*)
ASHB(:= sop = 1066_8) → (CC,Db ← Db × 2^s);	(**byte arithmetic shift*)
NOR(:= sop = 067_8) → (CC,D ← normalize(D)); (R[r′] ← normalize␣exponent(D));	(**normalize*)
NORD(:= sop = 1067_8) → (Db ← normalize(Dd); R[r′] ← normalize␣exponent(D));	(**normalize double*)
SWAB(:= sop = 3) → (CC,D ← D⟨7:0, 15:8⟩)	(*swap bytes*)

Logical Operations

BIC(:= bop = 0100) → (CC,D ← D ← D ∧ ¬S);	(*bit clear*)
BICB(:= bop = 1100) → (CC,Db ← Db ∨ ¬Sb);	(*byte bit clear*)
BIS(:= bop = 0101) → (CC,D ← D ∨ S);	(*bit set*)
BISB(:= bop = 1101) → (CC,Db ← Db ∨ Sb);	(*byte bit set*)
BIT(:= bop = 0011) → (CC ← D ∧ S);	(*bit test under mask*)
BITB(:= bop = 1011) → (CC ← Db ∧ Sb);	(*byte bit test under mask*)

```
Branches and Subroutines Calling: PC ← f;
  JMP(:= sop = 0001₈) → (PC ← D');                                  (jump unconditional)
  BR(:= brop = 01₁₆) → (PC ← PC + offset);                          (branch unconditional)
  BEQ(:= brop = 03₁₆) → (Z → (PC ← PC + offset));                   (equal to zero)
  BNE(:= brop = 02₁₆) → (¬Z → (PC ← PC + offset));                  (not equal to zero)
  BLT(:= brop = 05₁₆) → (N ⊕ V → (PC ← PC + offset));               (less than (zero))
  BGE(:= brop = 04₁₆) → (N ≡ V → (PC ← PC + offset));               (greater than or equal (zero))
  BLE(:= brop = 07₁₆) → (Z ∨ (N ⊕ V) → (PC ← PC + offset));         (less than or equal (zero))
  BGT(:= brop = 06₁₆) → (¬(Z ∨ (N ⊕ V)) → (PC ← PC + offset));      (less greater than (zero))
  BCS/BHIS(:= brop = 87₁₆) → (C → (PC ← PC + offset));              (carry set; higher or same (unsigned))

  BCC/BLO(:= brop = 86₁₆) → (¬C → (PC ← PC + offset));              (carry clear; lower (unsigned))
  BLOS(:= brop = 83₁₆) → (C ∧ Z → (PC ← PC + offset));              (lower or same (unsigned))
  BHI(:= brop = 82₁₆) → ((¬C ∨ Z) → (PC ← PC + offset));            (higher than (unsigned))
  BVS(:= brop = 85₁₆) → (V → (PC ← PC + offset));                   (overflow)
  BVC(:= brop = 84₁₆) → (¬V → (PC ← PC + offset));                  (no overflow)
  BMT(:= brop = 81₁₆) → (N → (PC ← PC + offset));                   (minus)
  BPL(:= brop = 80₁₆) → (¬N → (PC ← PC + offset));                  (plus)
  JSR(:= sop = 004Ø₈) → (                                           (jump to subroutine by putting
    SP ← SP − 2; next                                                R[sr], PC on stack and loading
    M[SP] ← R[sr];                                                   R[sr] with PC, and going to sub-
    R[sr] ← PC;                                                      routine at D)
      PC ← D);
  RTS(:= i = 00020Ø₈) → (                                           (return from subroutine)
    PC ← R[dr];
    R[dr] ← M[SP];
    SP ← SP + 2);

Miscellaneous processor state modification:

  RTI(:= i = 2₈) → (PC ← M[SP];                                     (return from interrupt)
                    SP ← SP + 2; next
                    PS ← M[SP];
                    SP ← SP + 2);
  HALT(:= i = 0) → (Run ← 0);
  WAIT(:= i = 1) → (Wait ← 1);
  TRAP(:= i = 3) → (SP ← SP + 2; next                               (trap to M[34₈] store status and
                    M[SP] ← PS;                                      PC)
                    SP ← SP + 2; next
                    M[SP] ← PC;
                    PC ← M[34₈];                                    (enter new process)
                    PS ← M[12]);
  EMT(:= brop − 82₁₆) → (                                           (emulator trap)
                         SP ← SP + 2; next
                         M[SP] ← PS;
                         SP ← SP + 2; next
                         M[SP] ← PC;
                         PC ← M[30₈];
                         PS ← M[32₈]);
IOT(:= i = 4) → (see TRAP)                                          (I/O trap to M[20₈])
  RESET(:= i = 5) → (not described)                                 (reset to external devices)
  OPERATE(:= i⟨5:15⟩ = 5) → (                                       (condition code operate)
    i⟨4⟩ → (CC ← CC ∨ i⟨3:0⟩);                                      (set codes)
    ¬i⟨4⟩ → (CC ← CC ∧ ¬ i⟨3:0⟩)));                                 (clear codes)
                                    end Instruction␣execution
```

A systems approach to minicomputer I/O

by FRED F. COURY

Hewlett-Packard Company
Cupertino, California

INTRODUCTION

You can tell a lot about a guy by the way he draws a block diagram of a computer system. If he draws the central processor and memory as small boxes off in a corner, then proceeds to fill the page with an elaborate portrait of the input/output system, he is usually referred to as (among other things) an "I/O type".

I have drawn several such diagrams, and I offer this information as a caveat to the reader.

In the pages to follow, I shall outline and attempt to justify some of my views on minicomputer I/O, particularly on "where we should be going from here". If some of the suggestions are already being implemented, I think they are steps in the right direction. If, on the other hand, some of the ideas seem too far out, consider the source.

A BIT OF HISTORY

I guess things started the way they did for several reasons. Hardware (relays, vacuum tubes, power supplies, and air conditioners) was very expensive, especially in the large quantities necessary for computing. The resulting machines were so incredibly complex (literally thousands of relays and vacuum tubes) that just getting one to work was a major accomplishment. In spite of the complexity involved, the actual capability of the early machines was limited to large-scale automatic number-crunching.

It is not hard to understand that hardware optimization was foremost in the designers mind. Unfortunately, programming these first machines was quite difficult due to the limited storage available in the machines, and also due to the fact that no programming frills (such as assemblers) were provided.

I/O was no real problem, since most of the early machines were clearly compute-bound, especially in number-crunching applications, and most I/O was simple card input, line printer (or card) output.

Engineers took advantage of technological developments (core memories, transistors) to build faster, more powerful machines. Programmers began to apply the new machines to a wide variety of problems (such as writing assemblers) and began to explore the true potential of computers.

As the number of machines increased, users (programmers) began to outnumber designers (engineers). They wanted to have something to say about the design of the machines they would be using *before* it was too late.

The engineers made the computers work, but the programmers made the computers *do* something. It was recognized that the important parameter to optimize was overall system performance. The engineers had to worry not only about how fast a machine could multiply two numbers together, but how efficiently the machine could be programmed to invert a matrix.

It is now common practice for computers to be designed by teams of engineers (with programming experience) and systems programmers (with hardware understanding) in order to optimize the overall performance of the resulting hardware/software system.

Also, the emphasis is shifting from hardware minimization to people optimization. As the cost of hardware goes down, and the cost of people goes up, the way to minimize cost is to maximize the efficiency of people in the design, production, programming, and eventual use of the system.

A GLIMPSE INTO THE FUTURE

In the near future, especially in some of the new minicomputer markets, the vast majority of computer users will not be programmers. As a matter of fact *these users will not want to program computers.* They won't particularly even want to use computers. They will have questions to be answered, problems to be solved, and things to be done. If a computer offers a better way (or, in some case, the only way) to do it,

people will consider using a computer. Otherwise, they will choose another method, or not do it at all.

Let's face it . . . the novelty is wearing off. The small computer industry must come of age. We are approaching the same position as the commercial airlines are in now. People don't fly just because they want a plane ride. *They want to get somewhere,* and flying happens to be the best (fastest, cheapest, most convenient) way to get there. If it's not, they will choose a better way to go or they will stay at home.

And most people are no longer interested in "roughing it" (wearing goggles and helping to start the engines). In most cases, the less they are aware of the fact that they are flying, the better they like it. This attitude is reflected in boarding ramps at the airports, and music, drinks, dinners, and movies while in flight.

And people are only interested in new developments insofar as *they* are directly affected. A revolutionary new jet aircraft design is of interest only if it means a faster, quieter, or more comfortable trip. Note what is stressed in the Boeing 747 advertisements. New navigation, propulsion, and control systems are ignored in favor of winding staircases and plush accommodations. Pilots fly planes, people pay to ride in them, and there are a lot more people than pilots.

The same rules will apply to minicomputers. New architectures, bussing structures, and addressing modes are only appreciated in terms of benefits which the user can see. Applications programs will be written *for* the user, not *by* him, and he will only be interested in the performance of the entire system as it affects his particular problem.

THE MYTH OF THE ULTIMATE PROCESSOR

But we are continually improving our machines. We are coming up with better performing hardware/software systems every day.

I don't think that faster processors and more powerful languages are the whole solution. Let me illustrate by carrying the current trends to their ultimate goal.

Suppose a man wants to generate an amortization schedule for a home loan. State of the art in minicomputers has reached the point where he can get a zero-cost infinitely-fast processor with 4K of memory and a super-powerful new compiler called "ENGLISH". The steps he goes through to generate the amortization schedule may be familiar to many readers:

1. He sits down to tell the computer (in "ENGLISH") to generate a loan amortization schedule. He discovers that no I/O device was provided. So he buys a teletype (with controller) for $2,000.
2. He tries to load the "ENGLISH" compiler paper tape into the machine. Discovers that "ENGLISH" requires 8K of core; he only has 4K. So he buys another 4K of core for $5,000.
3. He is about to load "ENGLISH" when he discovers that the MTBF on the teletype is shorter than the time it takes to load the tape. So he buys a high-speed photoelectric paper tape reader for $3,000.
4. He loads the "ENGLISH" compiler.
5. He types (in "ENGLISH") "GENERATE AMORTIZATION SCHEDULE (CR, LF)"
6. Immediately, the system starts to punch a binary tape. However, halfway through, the teletype punch breaks down. So he buys a high speed punch for $2,000.
7. He punches the binary tape.
8. He loads the binary tape.
9. He starts the program and types in the amount of loan, interest rate, and term.
10. Immediately, the system starts printing output, one line for each monthly payment. It takes a total of forty-five minutes to print all 360 lines. Meanwhile, the man stands there, with his fingers in his ears, hoping that the teletype printer will not break down before all the output has been printed.

The following chart compares the price/performance characteristics at the beginning and at the end of the example:

	Before	*After*
Price	$0	$12,000
Speed	Infinite	10 char/sec.

Some may say that the example is an exaggeration. It may be, but I wonder if they have ever tried to generate an amortization schedule using a minicomputer in its "basic configuration".

The point is, that if one were to substitute zero-cost, infinitely-fast processors into most existing minicomputer systems, the total system cost and overall system throughput would not be significantly affected.

A CALL FOR UNITY

So far, I have tried to make three points:

1. Computers should be designed for the user, not for the designers. The user wants a system to solve his problems, not a computer to program.
2. The best way to optimize the overall performance of a system is to take a unified approach in the design of the system's components in order to optimize their performance together.

3. I/O is by far the weakest link in current minicomputers. The total cost and overall performance of most existing minicomputer systems would not be greatly affected if we substituted a zero-cost, infinitely-fast processor and a super-powerful programming language.

The conclusion I draw from these points is that if we are to improve the overall performance of minicomputers, we must concentrate more on I/O. However, I don't think that faster, cheaper I/O devices are the whole answer. There is no question that we need such devices, but we need something more.

We need to include I/O in the design process from preliminary specification through actual construction. I/O is an integral part of system performance and it should be an integral part of the design process. Processor architecture, instruction set, and I/O scheme should be developed together, from scratch, in order to truly optimize total system performance.

I don't think we should discuss minicomputer I/O as an isolated topic; rather it should be treated as an integral part of the whole system. As soon as we look at I/O in this light, several very interesting possibilities appear.

A BIT OF PHILOSOPHY

Before we approach the problem of new I/O schemes, let us approach the problem of approaching problems. I think that we often misdirect our efforts due to taking too narrow a view of a given problem. It's like struggling to climb over a wall when, if we had stepped back and looked at the whole scene, we would have seen the open gate a short distance away.

The important thing is to define the *real* problem (in this case, to get to the other side of the wall, *not* to climb over the wall) and to take a sufficiently broad view of the problem so as to include several alternative paths from which to select the best.

Don't look for a way to improve existing methods. Rather, carefully define the *real* problem, then try to find the best way to solve that problem. The best solution may be to improve upon existing methods, but then it may be a totally different approach.

Rapid advances in technology necessitate constant reevaluation of goals and methods. Decisions which were valid two years ago may have lost their validity due to technological developments.

Let us try to reanalyze some of the basic characteristics of I/O and perhaps suggest some new approaches to minicomputer I/O design in the light of current (and projected) technology

A VERY BASIC DISTINCTION

I/O operations can be divided into two groups:

1. Those which are *intrinsic* to the solution of the problem at hand, and
2. Those which are *incidental* to the solution.

Let us analyze the loan amortization problem discussed earlier, and classify the I/O operations performed according to the above criteria. The problem, as you recall, was to generate a loan amortization schedule (*not* to program a computer to generate the schedule. The difference here is important as will be seen).

The I/O operations involved are classified as follows:

Intrinsic

1. Input loan description
2. Output amortization schedule

Incidental

1. Load compiler
2. Type program
3. Punch object tape
4. Load program

Note that the division would have been quite different if the problem had been defined in terms of programming a computer to generate the schedule. Unfortunately, we "computer types" have grown so accustomed to this rigmarole that we accept it as a part of problem solving. It is difficult for us to distinguish between the two because we are so used to working with machines.

(If you have a hard time categorizing the I/O steps in a particular application, try describing the sequence of operations to your wife. Those operations which she accepts and understands without further explanation are intrinsic, the others are incidental.)

The goal of new I/O design approaches should be to streamline the intrinsics and to eliminate the incidentals. If an incidental operation cannot be eliminated, it should be made transparent or at least as painless as possible.

COMPUTERS TALK TO PEOPLE

Until recently, man/minicomputer communications have been rather poor. The teletype has been by far the predominate minicomputer I/O device, primarily due to an unapproachably low cost for a combined keyboard, printer, tape punch, and tape reader facility.

Rather than ask how we can improve upon the teletype, let us ask "What is the best way to talk to

computers?" The answer is contained in the question. Most interpersonal information is conveyed by speech. Even "HAL", the ultimate computer talked and listened to people. ("Yes", you might say, "but look what happened to him.") Notice, however, that not a single teletype was to be seen (or heard) throughout the entire Space Odyssey.

Unfortunately, inexpensive spoken communication with minicomputers is not (yet) within the state of the art. So we must ask what is the next best method. Obviously it is visual communications.

Man can assimulate visual information very rapidly. Ten characters per second is much too slow, one hundred per second is adequate, and a picture is worth a thousand and twenty-four words. I think that we are on the right track with some of the low-cost CRT terminals which have been and are being developed. One objection which is usually raised about CRT output is the lack of hard copy. True, this may be a limitation in some instances, but how often do you *really* need hard copy? Suppose you could store scrolls of output in a file somewhere and call them back for CRT display and manipulation very rapidly? Again, the solution space is different for different statements of a problem.

Now, how should man talk to a computer? Remember, most new users will be non-technically oriented. We should attempt to tailor the computer to the people, not vice versa. Let the machine do the work. This is in keeping with the trend toward less expensive machines and more expensive people.

I firmly believe that the human finger is much better for pointing than for typing. Given a fast CRT output, a very efficient input method is the selection of a reply from a computer-generated "menu". Let the computer guide the user and help, rather than hinder, in the solution of his problems.

COMPUTERS ALSO TALK TO MACHINES

Peripheral device interfacing is the area where we have had more experience, since we have long been attacking such problems as "How can we make our machine talk to a teletype?" (instead of "How can we make our machine talk to the person sitting at the teletype?").

I think new developments in technology and new applications areas warrant a new look at the area of interfacing peripheral devices to minicomputers. I think we can find ways to design better device controllers, faster and at a much lower cost.

We spend most of our time trying to develop integrated processor/software systems. We take advantage of quantity production techniques to lower hardware costs. We do everything we can to minimize engineering and programming time for the basic system, then we design a unique controller (and write new support software) for each new peripheral device.

We are very interested in statistics concerning the amount of time our CPU's are busy, but do we realize how inefficiently our device controllers are used? Most integrated circuit devices can easily run at a ten megacycle clock rate. Yet an I.C. teletype interface runs on a 110 cps clock. A typical photoelectric reader reads 300 characters per second. This means that such device controllers are only active on the order of 0.001 percent of the time. The remaining 99.999 percent of the time, the high speed logic gates are idle and only a few flip-flops are needed to hold some logical state information.

To me, this clearly suggests multiplexing, or in some way time-sharing the control logic among several devices.

PARTITIONING OF I/O FUNCTIONS

The inclusion of I/O design as an intrinsic part of the overall computer system design provides a much larger space over which to distribute the functions necessary for I/O operations.

For example, we could choose to implement a full duplex teletype controller using only one flip-flop, a clock, and two level converters, and provide timing and control functions in software.

To add a photoreader merely requires device addressing capability, perhaps another flip-flop, and an addition to the software.

This argument begins to fall apart when we add too many devices, or hang on a fast device (such as a magnetic tape unit). But does it really fall apart? How much I/O could your minicomputer handle if all it had to do was I/O? How much more could it handle if the I/O routines were in read-only-memory, rather than in core? Minicomputers are commonly being used to handle I/O for larger machines.

"But", you say, "separate I/O processors are only warranted for very large machines. They are much too expensive to be included in a minicomputer."

A WAY-OUT IDEA(?)

Let us consider this approach before we dismiss it as unrealistic. Suppose we were designing a minicomputer as an integrated CPU/software/I/O system. We could

choose to include two identical sets of processor logic, each with access to main memory and to each other. We could micro program one to act as a CPU and the other as an I/O processor. We could provide the absolute minimum hardware necessary in the device controllers and let the I/O processor do the rest of the work.

Would this really be expensive? How much would it cost to add a duplicate set of cards? They have already been designed. Provisions have been made for their production and testing. Programming support has long since been developed. The existing core memory, power supplies, and cabinet can be shared.

The amount saved in device controller design and implementation should greatly exceed that spent for the I/O processor.

And consider the power of such a system. I/O instructions, block data transfers, virtual memory schemes, multi-level priority interrupts, and special user-defined I/O functions all take on a new dimension.

Whether or not such an I/O scheme is feasible requires much further consideration. The important point is that it is an example of what might be possible in an integrated design approach.

SUMMARY

1. Minicomputer performance and development is I/O bound, especially in the man/machine interface area.
2. It is time we stood back and looked at minicomputer I/O, not in terms of how we can improve on existing techniques, but by analyzing what we want to do (the total problem) and deciding on the best way to do it (a total solution).
3. I/O should be designed *into* not *onto* the system. An integrated CPU/software/I/O design will result in optimum performance.
4. New approaches to existing problems might lead us in exciting new directions.

A multiprogramming, virtual memory system for a small computer

by C. CHRISTENSEN and A. D. HAUSE

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

OVERVIEW

The specific objective of this small computer system is to interface six to eight small graphical terminals[1] to a large batch-processing computer. The small computer provides the graphical terminals with real-time processing for generating, editing and manipulating graphical or text files. The small computer passes along to the large computer requests for large tasks. Access to the data base in the large computer is provided. Another aspect of this objective is remote concentration. The terminals are connected to the small computer directly or through several DATA-PHONE® 103 data sets. The small computer is connected to the large computer through a single DATA-PHONE® 201 data set. This configuration reduces communication costs for a group of terminals located remotely from the large computation center.

The general objective of this system is to investigate memory management strategies in small computers. In particular, can large computer techniques be applied? How big are the required programs? To what extent can high processor speed be substituted for large primary memory size?

The hardware configuration for the system is as follows: The computer is a Honeywell DDP-516 with an 8192-word, 16-bit, .96 μs core memory. Secondary memory is a special fixed-head disk by Data-Disk, Inc., which has 64 tracks, packed 8192 words/track, and operates at 30 rps rotational velocity. The disk is connected to the computer through a high-speed Direct Memory Access Channel. The disk is sectored in 8-word blocks (hence, a 16-bit address just suffices for the disk sectors). A Soroban 600 cpm card reader is connected to the computer I/O bus. A special serial transmission system (called the I/O loop) is also interfaced to the I/O bus. Currently, four DATA-PHONE® 103 data sets are connected to the I/O loop, but other I/O hardware may be added relatively easily.

Low-level software support (primarily, a fancy MACRO assembler) is provided on a GE-635 computer, not on the DDP-516 itself. Loaders and debugging aids have been written for the DDP-516.

The system supports a virtual-memory addressing scheme and a multiprogramming user environment.[2] The system manages memory (moves programs and data between core and disk, on demand) and all disk I/O, and provides the low-level interrupt handler for the local teletypewriter, the card reader, and the I/O loop. The system supports virtual addressing by providing a mechanism to convert virtual addresses to real core addresses, a task that requires memory management if the addressed data is not currently in core. The multiprogramming support is provided in the form of the tables and memory management required to automatically switch control from one user to another without interference.

MEMORY MANAGEMENT

Segmentation

In order to free the programmer from the task of memory management, that is, the supervision of the movement of data and programs between primary (core) and secondary (disk) memories, it was decided that the programmer should address a large virtual memory space rather than the physical storage media. The system handles the tasks of converting virtual addresses into physical addresses and of making the data available for processing (moving data into core). Space is allocated within the virtual memory on the basis of segments.[2] As usual, a segment is a named block of storage which contains contiguous words. The first word of a segment is at relative address 0. In our system, a segment may contain any number of words up to 2047. Furthermore, the segment is the physical

storage allocation unit; segments are not physically subdivided (paged). A physical segment consists of a logical segment plus a few extra words required by the memory management system.

Although segments are permitted to contain 2000 words, in practice most segments are limited to 512 or fewer words. One reason for this limitation is that a DDP-516 memory-reference instruction has a 9-bit address field, so that a reference to a location greater than 511 would require indirect addressing. Moreover, the approximately 4000 words of core storage available for segments would be too easily clogged by larger segments.

For convenience, ID's (segment numbers) rather than segment names are used internally for segment references. An ID is assigned the first time the segment is encountered, e.g., when it is first loaded into the system, or when a data segment is created at run time. Whenever a segment is deleted, its ID is reclaimed for future use. The ID is a 15-bit number, which means that the system can handle in excess of 32,000 segments. Hence, if each segment contained 500 words, the virtual memory space would contain 16,000,000 words, far in excess of our present physical storage capacity. As a practical matter, the current implementation permits just 4096 ID numbers, but this can easily be expanded if required.

Files

In order to facilitate handling large character strings that would not fit into a convenient size segment, we have implemented a higher-level storage classification called the file. A file is a linked group of segments. Each segment in a file has a forward and backward pointer ID to the succeeding or preceding segment in the string. A user has a private file directory which lists his own files by name. The directory provides a link from file name into the file via the ID of the first segment in the file. Note that files may be any size, because there is no imposed limit on the number of segments that may be linked together, other than the total number of ID's available.

The system provides routines for accessing files by name and for fetching and storing characters in a file. These routines make the segment boundaries invisible to the user, which is a great programming convenience.

The system also provides a public file directory so that data may be shared. One user's private files are inaccessible to other users.

Addressing

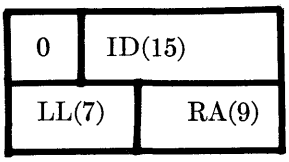
The hard-core system program uses conventional DDP-516 addressing, without any special software structure or restrictions. Since the hard-core system is "bolted in," the memory management system does not have to handle it. The segmented programs (those which are handled by the memory manager and use the virtual memory space mentioned previously) require four specialized addressing modes for various purposes. Before these modes are described, it would be helpful to consider two major system requirements that the addressing scheme was designed to support. First, it was required that the memory manager not have to relocate addresses within a segment when that segment moved from one core location to another. This requirement could be removed, but it would significantly increase system overhead (both space and time). Second, it was required that segmented programs not contain internal variable storage, so that such segments could be used concurrently by any number of users without interference. Thus, it is not necessary to provide multiple copies of such "pure procedure" segments, which saves core storage space.

The first specialized address to be considered is the intra-segment pointer, i.e., an address that points to a location within the same segment that contains the pointer. Fortunately, it was possible to satisfy the relocation requirement by pre-empting the DDP-516 index (X) register for use as a base register. This scheme works because the index register can be attached to any address, whether it be in a memory-reference instruction or a full-word indirect address. Furthermore, indexing is controlled independently of indirect addressing. In particular, whenever a segment is in execution, the index register contains the starting address of that segment. Then, all intra-segment references have the index bit "on" and the address field set to the desired relative address within the segment.

The second specialized address is the absolute address. As its name implies, this address points to a fixed core location in sector 0 (otherwise, such an address could not be used in a memory-reference instruction). Hence, the index bit is 0. There are two distinct uses for the absolute address. One use is to refer to fixed information in the system. In particular, a transfer vector is required in order to reach various system subroutines, none of which are located in sector 0. In addition, a pool of generally useful constants is provided so that segments may be spared the necessity of containing their own copies of these constants. The other distinct use of sector 0 is to provide a pool of temporary storage locations for use by segmented programs. This pool of variables (the "thread-save") belongs to the currently executing thread. The thread-saves of all other existing threads occupy other places within core. When another thread is given control, its

pool must be moved into sector 0. It should be noted in passing that if our computer had possessed a second index or base register, we would have used it to point to the thread-save and thus avoided the need to physically swap thread-save data when switching threads.

The third specialized address is a software-interpreted address called the virtual address. The virtual address is the general inter-segment address. It consists of two words. The first word contains the ID of the

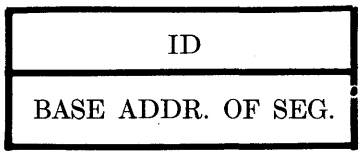


desired segment. The second word contains two fields. The low-order 9 bits (RA) of the second word is the relative address of the word within the segment. Note that only the first 512 words of a long segment can be referenced, which matches the limitation of memory-reference instructions. The left 7 bits of the second word (LL) form the "loose link," which is a pointer into the 128-entry table of in-core segments. The loose link need not point to the correct entry in the table, but if it does, conversion from virtual to physical core address is relatively rapid (~25 microseconds). Hence, whenever the system is required to convert a virtual address into a physical address, the loose link is properly set, so that subsequent address conversions will go at maximum speed. Note that the loose link contradicts the "no variable storage within a segment" requirement, because the system can change the loose link. However, such a change is never harmful, because the segment table entry is checked before it is used. Also, since the same segment table is used for all threads, a correct loose link for one thread will be correct for any thread.

The fourth specialized address is a direct address. A direct address is an absolute core address of a location inside a segment. A direct address must be stored within a certain subset of the thread save, in order that the system be able to find and relocate the address if the segment is shifted to a new core location. Moreover, a segment that is referenced by a direct address is locked into core, else use of the direct address could give a spurious result. The direct address does not add any significantly new addressing capability beyond that provided by the virtual address. However, it is useful because it is fast (hardware interpretation) compared to the virtual address. Use of the direct address does inflict some costs, in particular, the locking into core of the referenced segment, and the system overhead required to relocate the direct addresses during a core shift.

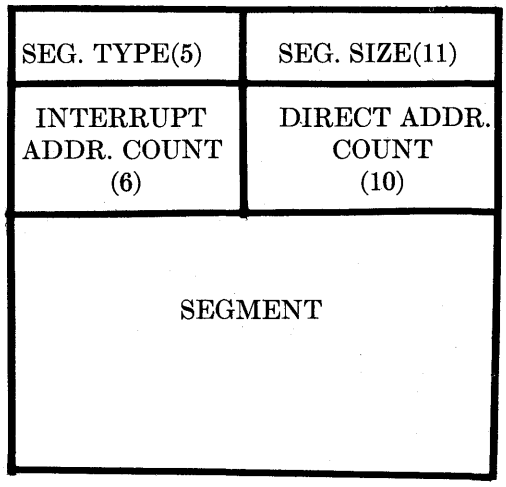
CORE MANAGEMENT

The 8192-word memory of the DDP-516 is divided into two approximately equal parts, the hard-core system and segment storage. The hard-core system is the portion of the system that resides permanently in core memory. The rest of memory is allocated in variable size blocks to segments, as required. Each in-core segment is accessed through its entry in the segment table. A segment table entry is composed of two words, the segment ID and the segment location in core.



A virtual address (ID, RA) is converted to an absolute core address by adding the relative address (RA) to the base address of the segment (second word of segment table entry). If the ID of the desired segment cannot be found in the segment table, then the segment is not in core and must be fetched from disk.

When core is filled with segments and a new segment is required, one or more of the in-core segments must be pushed (written onto disk or just discarded) to make room for the new segment. The algorithm for choosing which segments to push out of core is simple. A sequential scan of the segment table produces push candidates. The scan begins where the last push scan ended and ends when successful pushes have yielded the desired amount of space. A candidate is pushed if and only if the second of the two segment header words contains, zero, i.e., there are no direct or inter-



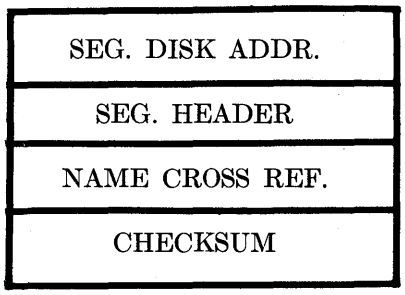
rupt addresses pointing to the segment. The leading bit of the first header word (hence, the leading bit of the segment type) tells whether the segment must be written on disk when it is pushed.

Each time the user establishes a direct address to a segment, the direct address count for that segment is incremented. When the direct address is deleted the count is decremented. The direct addresses stored on the users' call push down lists are also included in this count. When a call is executed a direct address to the called segment is pushed on the list and the count for the called segment is incremented. Upon execution of a return the direct address is popped off the list and the count is decremented. Hence, all segments on user push down lists are locked into core, as are all segments pointed to by direct addresses established by the user. I/O interrupt handlers are also allowed to be segments; when in use they are locked into core by incrementing the interrupt address count (high order six bits of the second segment header word).

When enough segments have been pushed out of core to make the desired amount of space, the holes left by the pushed out segments are gathered at the top of core. This is accomplished by moving all the segments above the holes down over the holes. Moving the segments in core requires that all direct addresses be changed to reflect the core shift. These include the segment addresses in the segment table, the users' call push down lists, direct addresses established by the users and various other system pointers. Since this is a long list and shifting the segments down is a long task (about 50 milliseconds), an attempt is made to free a large block of space (currently 1000 words) instead of just the amount requested. This makes the next few space requests easier to fill since a segment push and core shift are not required.

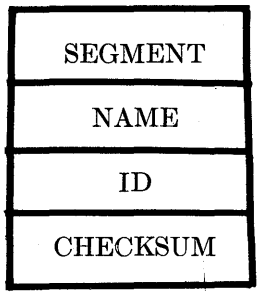
DISK MANAGEMENT

The half-million-word disk attached to the DDP-516 is divided into three areas. Eight K (K = 1024 words) is reserved for saving and restoring the hard-core system, 32K is allotted to disk management tables and the remaining 472K is used to store segments that have been pushed out of core. These segments on disk can be accessed by name or ID using the disk name table or disk ID table. Both tables are themselves on disk and comprise the disk management table area. Each entry of the disk ID table contains four words.



When a segment on disk is accessed using its ID the corresponding disk ID table entry is read and the segment's size is extracted from the segment header word (2nd word of entry). Then the system makes sufficient space in core for the segment. Next the segment's disk address (1st word of entry) together with its size and the starting core address of the space just acquired are given to a routine that reads the segment into core.

Each entry of the disk name table also contains four words.



When a segment is accessed using its name, the corresponding disk name table entry is accessed with the aid of a hash coding technique.[3] The ID (3rd word) is then used to access the segment as previously described. The cross-referencing words in the two disk tables facilitate conversion from name to ID or ID to name.

When a segment is pulled into core from disk a flag bit in the segment's header word called the disk restore bit is reset. If any changes are made to the segment while it is in core, this bit is set to 1, which indicates that the segment should be rewritten on disk if it is pushed out of core. Otherwise, the segment is thrown away when pushed, since an up-to-date copy already exists on disk. A new segment is created with the disk-restore bit set; when it is pushed out of core, a zero disk address in its disk ID table entry indicates that space for it must first be allocated on the disk. Segments can also change size while in core. If a segment is too large for its old slot on disk a new disk slot is

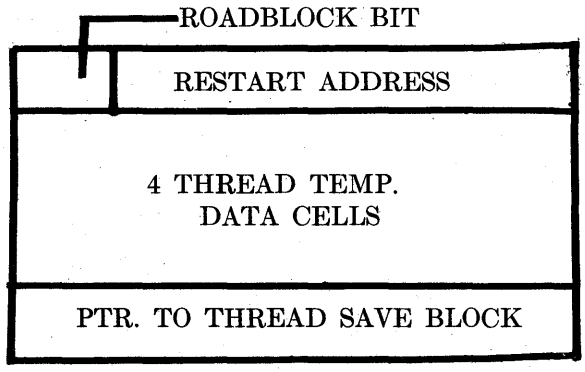
allocated and the old one is marked as a hole to be collected later.

Disk garbage collection is triggered when the disk allocation pointer approaches the end of disk storage. An autonomous thread is then initiated in the multiprogramming system which relocates the segments on disk, bubbling the holes to the top. This is a long procedure and executes concurrently with other threads.

MULTIPROGRAMMING

The multiprogramming system uses a pure roadblock strategy, i.e., it gives a thread control and lets it compute until it roadblocks. The next thread is then given control until it roadblocks, etc. There is no fixed maximum slice of time for each thread. A thread can roadblock for several reasons. If the thread requests input or output a roadblock occurs and the I/O proceeds under interrupt control. When the I/O is complete the thread is unroadblocked. A thread can also address a segment which is not in core and roadblock until the segment is brought in from disk. If a thread is still roadblocked (I/O not completed yet) when its turn comes around again it will be skipped. Thus a thread is given control only when its roadblock is removed and its turn comes around.

The heart of the multiprogramming system is the thread table. It contains a six-word entry for each of the possible ten threads.



When the system is otherwise idle it scans this table for an unroadblocked thread (roadblock bit = 0). When a thread does unroadblock, its four temporary data cells are transferred to core sector zero, and the thread is restarted at the address specified by the first word of the thread table entry. This restart address is always within the hard-core system; before control is passed to an outside program segment the thread's save block is moved into core sector zero. The thread save block is 80 words long and resides in the segment storage part of core. However, it is never pushed out of core.

THREAD SAVE BLOCK FUNCTION	NO. OF WORDS
PUSH DOWN LISTS	24
USER TEMP DATA	16
USED DIRECT ADDR.	8
SYS. TEMP DATA	8
SYS. DIRECT ADDR.	15
MISC. SYS. POINTERS AND DATA	9

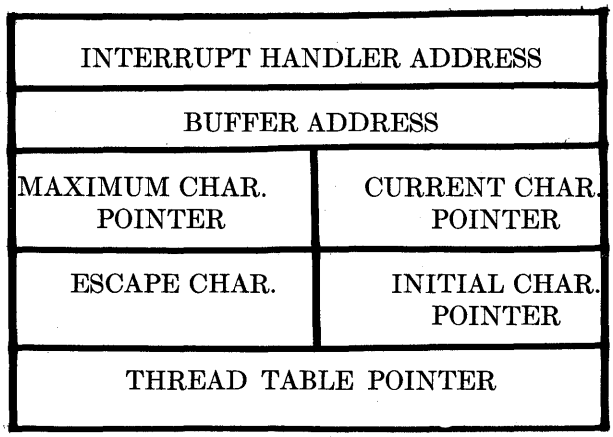
The thread save block contains all the data and pointers required by system and user in order to implement pure procedure programs. Before a thread's save block is moved into core sector zero the save block in core sector zero is restored to the previous thread's save block. This data movement constitutes most of the overhead involved in changing threads and takes about a millisecond. However, most of the roadblocks that occur when a thread is using hard core system programs require only the four data cells in the thread table entry to be in core sector zero. This brings the thread changing time down to about 50 μsec. For example, when an out-of-core segment is addressed a thread could be roadblocked three or four times for things like reading the disk ID table, making space for the segment, and finally transferring the segment into core. Only after the segment is in core and the address is about to be computed is the complete thread save block required to be in sector zero.

There are several other interesting roadblocks which can occur. For example, a low usage program may be more compact and simpler if it is not pure procedure (required by multiprogramming). This is allowed by using a GATE statement at the start of the program. The gate allows only one thread to be in the program. Any other threads that tried to enter would be roadblocked until the first thread opens the gate on its way out. Of course, allowing only one thread at a time rules out this technique for high usage programs. A program can also request a thread to give up control. This is a useful technique to break up programs with long execution times or to wait for some external event to occur without locking the other threads out of the machine.

INPUT/OUTPUT

All input/output in the computer is done under the interrupt system with the aid of the I/O table. This

table contains a five-word entry for every I/O device attached to the computer.



The above example is a table entry for a character-oriented device. When input or output are desired the appropriate program is called with the buffer segment and the escape character supplied as arguments. The called I/O program then fills in the I/O table entry, primes the I/O device and roadblocks the thread. When an interrupt occurs, the system gives control to the location specified by the interrupt handler address in the I/O table entry for the interrupting device. For input, the characters would then be inserted in the buffer using the buffer address and the current character pointer.

About 300 microseconds are required to handle each character interrupt for the teletypes. When an escape character match or full buffer is encountered the thread table pointer in the I/O table entry enables the program to unroadblock the thread, and the I/O function is complete.

Since a large number of different I/O devices are expected to be connected to the computer, with only a few of them active at one time, interrupt handlers are allowed to be program segments. When an I/O device becomes active its interrupt handler segment is fetched and locked into core for the duration of its activity.

Disk I/O is controlled by the disk I/O queue, which contains twenty entries of five words each.

DISK TRANSFER PROGRAM

DISK ADDRESS

CORE ADDRESS

THREAD TABLE POINTER

OTHER DATA

A disk transfer is initiated by finding an empty queue entry and inserting the address of the appropriate disk transfer program and its arguments. The requesting thread is then roadblocked.

The disk I/O handler is an autonomous process which goes on in the background of thread processing. When a disk I/O task is finished, the current disk rotational position is read and the disk addresses on the disk I/O queue are scanned to pick the task with the least latency. Control is then given to the picked entry's disk transfer program, which sets up the disk I/O. Upon completion of the task, the thread is unroadblocked using the thread table entry address in the queue entry.

USAGE

A brief description of currently available programs is provided in order to demonstrate some of the system capability. The system does not as yet serve a community of "outside" users or applications programmers.

Log-in

The teletypewriter log-in procedure is controlled by a segmented program. When a user dials the system, the interrupt handler answers the telephone and initiates a thread for the user. The new thread executes the log-in program, which requests a password. The password identifies a user catalog of files. Then the log-in programs sends thread execution to the monitor, which permits the user to select an applications program.

Text editor

One currently available applications program is a text editor. This is a very simple, line-oriented editor. It has the ability to enter one or more lines of text anywhere within an existing text file and to delete one or more existing lines. Of course, selected lines may also be printed out on the teletypewriter. The text input mechanism has a tab feature which permits the user to select the tab key as well as to position the tab stops. The command format is a single command letter followed by arguments, if any. The arguments are decimal line numbers for the print and delete commands, for example, or a line of text for the enter-text command.

The text editor may create or delete files, as well as attach any existing file. This capability is also pro-

vided by other programs, but it is convenient to have these features available from within the editor.

Interpreter

The second available applications program is a TRAC-like interpretive program.[4] This program can be used for text editing, but it is a general character-string manipulator. Like the editor, it is a "safe" program in that user errors do not bring down the system. Also, this interpreter provides a high-level programming language in which other applications programs can be written.

Debugging

We have provided a low-level debugging tool for use with segmented programs. This debugger is itself a segmented program, and it is designed to operate in the multiprogramming system environment. This is important, because it allows one user to debug without blocking other users. Also, it aids the programmer by freeing him from the necessity of knowing absolute addresses, which would be a painful requirement when the segments move within core or between core and disk.

The debugger allows the user to print out or alter the contents of selected locations within a segment. The user may also print or alter some of the data in his thread-save block. The command format consists of a one-letter command, followed by arguments, if needed. Numerical arguments are given in octal; segments may be referred to by ID or by name, as in the following address:

segname/ra

where segname is the segment name (in ASCII) or ID (in octal), and ra is the relative address (in octal).

SOFTWARE SUPPORT

It is becoming common practice to support small-computer programming on a large computer system. We have available a GE-635 computer in our computation center, and we use it to assemble all of our DDP-516 programs. The most important reason for using this kind of computation center support is that, by using the available assembler (GMAP), we have access to MACRO instructions and many powerful pseudo-operations that are not available in the manufacturer's assembler (DAP). The difference may not be important for small programs, but it makes a vast difference in a comparatively complex project, such as the current effort. Another significant advantage of computation-center assembly is that it uses convenient output peripheral equipment, in particular, card punches and line printers.

Our use of the large computer assembler to generate small computer code is not novel, but apparently it is not widely used. It works well in our case because the two computers (GE-635 and DDP-516) are organized similarly (e.g., they are both word-organized, single-address), and in a rough sense, the smaller computer is almost a "subset" of the larger computer. Moreover, GMAP has provision for redefining old instructions and for defining new ones. Hence, it is relatively easy to get GMAP to accept DDP-516 assembly code (in GMAP format, but with DAP mnemonics). The GMAP binary output cannot be changed so easily, however, so a separate program, called a post processor, has been written to convert the GMAP binary output into a more suitable format for our requirements.

For this project it was convenient to have two assembler/post processor packages, one for segmented programs, and one for system programs. The segment assembler helps the user with the special segment addressing modes, and simplifies access to system programs callable from segments. The segment post processor converts the resulting binary into a form suitable for use by the segment loader. The system assembler is simpler and less restrictive than the segment assembler, and it uses conventional (for GMAP) inter-program linkage features. The system post processor includes a linking, desectorizing loader which loads one or more relocatable programs into a core memory image, then punches the result as an absolute program which can be loaded into the DDP-516 by a simple, compact loader. In both cases, the post processor prints an octal listing of the final output as a debugging aid.

We have found it convenient to incorporate octal patch card facilities in both the segment loader and the system loader. Thus, the user is able to patch known errors in his binary decks before he has had a chance to reassemble.

STATUS REPORT

The system described above is currently working stand-alone (the 201 data set link to the large computer has not yet been implemented). The system supports four 103 data sets for communication with teletypewriter consoles. The graphical terminals are not yet available for connection to the system. Hence,

it is too soon to conclude whether the system can be used as a remote concentrator for graphical or other terminals connected to a large computer system. However, we can comment on the memory-management aspects of the objectives.

The current size of the hard-core system is just under four thousand words. This includes character handling routines, a 103 data set communication package, card reader package, in addition to the memory manager and multiprogramming support software. Hence, approximately four thousand words remain for program and data segments. The system has been exercised with four concurrent users; the segmented programs in use during this exercise included the text editor, the interpreter, and the segment debugger, which altogether represents five thousand words of program. The experiment generated and accessed several tens of thousands of data words. Delays due to multiprogramming were scarcely noticeable compared to delays due to disk latency. For example, it took ten seconds to sequentially access every character in a 20,000-character file. Note that this sequential access time is a function of the size of data segments that make up the file. The data segment for this experiment was 64 words in order to minimize the amount of core required by each user. Doubling the data segment size would halve the access time; it would also increase multiprogramming interference. More experiments are needed in order to explore such trade-offs.

REFERENCES

1 H S McDONALD W H NINKE D R WELLER
A direct-view CRT console for remote computing
Digest of Technical Papers International Solid-State Circuits Conference Vol 10 pp 68–69 1967

2 R C DALEY J B DENNIS
Virtual memory, processes, and sharing in MULTICS
Communications of the ACM Vol 11 No 5 pp 306–312 May 1968

3 R MORRIS
Scatter storage techniques
Communications of the ACM Vol 11 No 1 pp 38–44 January 1968

4 C N MOOERS
TRAC, A procedure-describing language for a reactive typewriter
Communications of the ACM Vol 9 No 3 pp 215–219 March 1966

5 D L MILLS
Multi-programming in a small-systems environment
The University of Michigan Technical Report 19 CONCOMP May 1969

Applications and implications of mini-computers

by C. B. NEWPORT

Honeywell, Inc.
Framingham, Massachusetts

Over the past four or five years the largest growth segment of the computer industry has been mini-computers and it appears that this trend will continue into the foreseeable future.

Mini-computers have typically been defined by their price rather than by performance. As recently as early in 1969, some observers were classifying mini-computers as those having a price for a minimum system of less than $50,000. Today a more reasonable figure would be $20,000 and some people may even press for $15,000 or even $10,000, but perhaps at this level one is talking of micro-computers.

These machines have had a fairly remarkable impact on the computer industry since in some respects their performance is even better than that of their big brothers which have built up the computer industry over the past 20 years. For instance, many computers have core cycle times and peripheral transfer rates which are considerably higher than the conventional large scale computers. Core cycle times of less than 1 microsecond are common and some machines are in the region of ¾ of a microsecond. Maximum I/O transfer rates are frequently determined solely by the memory speed and with 16 bit machines transfer rates of over 2 million characters per second are quite common.

It is interesting to compare these parameters with an IBM System 360/50 which has a core cycle time of 2 microseconds and a maximum I/O data rate of 800K bytes per second on the selector channel. The 360/50 could in no sense be classed as a mini-computer and of course in other areas such as core size, instruction repertoire, range of peripherals, standard software, etc., it is a far more powerful one than any mini-computer. Nevertheless this does indicate that for those applications where high speed minimum complexity processing is required, and rapid I/O transfers are needed, mini-computers may well be more effective than their large brothers.

Most large computers were designed basically for batch processing either scientific or business and the concept of high speed real time interaction with these machines tends to have been added as an afterthought. Thus, when one attempts to use large machines for systems such as air lines reservations, time sharing, messaging switching, industrial process control, etc., one finds that it is relatively easy to burden the large processor with the simple tasks of handling communication lines, attending to external interrupts, and interrogating large data files, thus leaving no time for the basic computation that may need to be done. The realization of this is leading to the off-loading of the simple jobs handled by large machines on to small peripheral machines (mini-computers) which can be dedicated to high speed but relatively simple tasks. Tasks include the handling of error control, polling, and conventional communication disciplines, over a number of communication lines, and then presenting packaged and checked messages to the large processor for subsequent handling.

In time sharing, it is becoming clear that the "number crunching" machine used to invert matrices, solve linear programming problems and so on, should be isolated from the relatively trivial tasks of sending and receiving messages to and from users and from the tasks of scheduling and monitoring the performance of the overall system. Mini-computers designed for high speed character manipulation rather than computation can undertake many of these housekeeping chores more effectively than the big machines.

In some instances mini-computers have proven to be very effective in taking on complete jobs that were previously thought to be the province of large machines. A good example is in message switching where mini-computers are showing that they can handle effectively the switching of messages between as many as 100 or 128 low speed communication lines. In this application essentially no calculation is required but there is an extensive amount of character manipulation, checking of character strings, and elaborate real time housekeeping. Figure 1 shows a diagram of a typical message

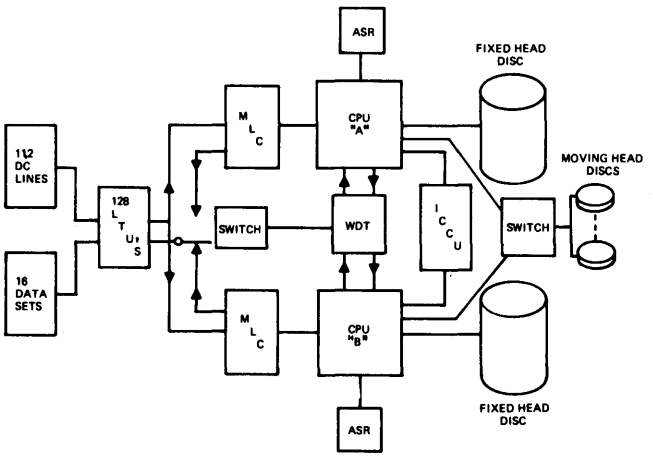


Figure 1—Message switching system

switch indicating the way in which two computers may be used to provide a switching function, and redundancy in the case of equipment failure. In a typical application, the CPUs would be DDP-516 class computers with 16K to 32K words of core, and a fixed head disc on each machine would be used for in-transit storage. The long term storage requirements for journaling and intercept would use moving head discs with replaceable disc packs.

The incoming communications lines, some having relay interface to dc lines, and others having EIA type interfaces to modems, would be connected into individual line termination units. These feed partially formed characters into the multi-line controllers (MLCs) which completely format the characters, check for parity, and perform other control functions before passing the characters and line identification into the CPUs. Input is taking place in parallel on both machines so that in normal operation they can both build up identical information on the in-transit disc stores. Communication between the CPUs is via the inter-computer communication unit (ICCU) and allows both machines to insure that they are in step on a message or partial message basis. The watchdog timer (WDT) is an independent hardware device monitoring the performance of both CPUs and providing an alarm and switch-over if one of the CPUs should fail. It will be seen that messages are inputted in parallel into both systems but outputted from only one of them.

Figure 2 shows a block diagram of the program which would be running in one of the processors. Input characters from the multi-line controller are passed through the input processor and assembled into partial message blocks in the input buffer area. These messages consisting of heading and text blocks are then passed to the disc queue and transferred from core to the fixed head disc. As messages are completed on the disc they are transferred back into core one at a time for header analysis and routing. This is undertaken by the message processing program, and completely processed messages are returned to the fixed head disc where they are queued ready for output to the appropriate line. As the lines become free the output processor program takes the messages off the fixed head disc, a block at a time, buffers them temporarily in core, and then transfers them to the multi-line controller.

It will be seen that the majority of the processing involved is examining strings of characters for particular sequences, and manipulating blocks of core storage being used for queues. Efficient data handling in both these areas and also the ability to operate with a high speed fixed head disc enables mini-computers to handle between one thousand and two thousand characters per second in a typical message switching application.

Mini-computers are normally quite limited in the amount of core storage they can have and it is interesting to note that in most communications applications there is a trade off between the amount of core storage required for input/output buffer blocks and queues, and the speed of the fixed head disc. With a high speed disc small buffer blocks can be used in core since these can be unloaded rapidly onto the disc before core saturation occurs. With a disc providing about 100 independent random accesses per second, buffer blocks holding in the region of 64 characters are normally acceptable and do not demand excessive core storage. However, if it were possible to increase the speed of the fixed head disc, by say 4 times, an approximately equivalent reduction in the size of the buffer blocks could be made and a corresponding reduction in the amount of core storage and hence in the cost of the system.

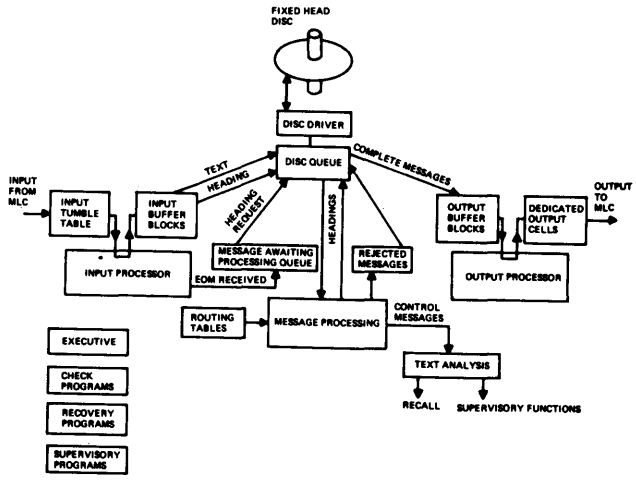


Figure 2—Diagram of basic message switching programs

It is interesting to note that there are essentially 3 different parts of the program: input processing, message processing and output processing. The communication between these 3 basic program segments is almost entirely through data held on the fixed head disc, with the exception of pointers and status words. This leads directly to the consideration of multi-processor systems to handle applications beyond the capability of one mini-computer. One computer would be assigned to each of the major processing areas and they would all have access to the common fixed head disc. Some simple means of passing limited amounts of status information between the computers would be necessary but all the major data flow would be thru the disc. The amount of core on each processor could be optimized to the task it had to perform and in principle so could the power of the processors, although in practice it would be simpler to maintain identical processors in all cases.

One would not expect the throughput to be as much as 3 times greater than that of a single processor because of the inability to share spare time between the processors. For instance, if the input is particularly heavy and the input processor is becoming overloaded it would be very difficult to arrange for, say, the output processor to take some of this load, whereas in a single computer system this can be arranged to happen automatically. While 3 computers might be expected to give somewhat less than 3 times the thruput of one computer, significant economies can be obtained in the redundancy since now one standby machine can be used to replace any one of the 3 normally operating computers. Thus 4 machines connected in the appropriate way can provide between 2 and 3 times the thruput of 2 machines connected in a normal redundant configuration. This type of configuration clearly gives savings in cost and an increase in reliability over the use of one or two large machines. In addition, it can simplify the programming and the checkout since separate tasks are confined to separate pieces of hardware.

Time sharing is another application area that has until recently been the province of large computers. There are, however, now on the market a number of small time sharing systems based on a single mini-computer. These systems typically provide for 16 or 20 users working with the interactive terminal language BASIC, or sometimes FORTRAN. These systems clearly do not compete with the larger time sharing systems in data storage, power and the varity of language facilities, library facilities, or ability to undertake extensive mathematical calculations. They do, however, provide a very useful service where simple fast access computation and data retrieval is required. The use of a multiple mini-computer configuration to extend the capability of the smaller systems upwards towards that

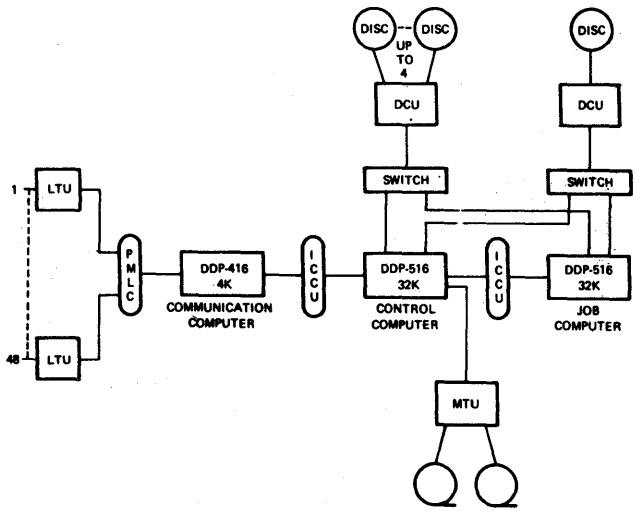


Figure 3—Time sharing system

of the large systems is well illustrated in the H1648 time sharing system. Figure 3 shows how the 3 computers are connected together to provide up to 48 simultaneous terminal users with the capability of programming in FORTRAN, BASIC, TEACH or SOLVE.

The terminals are handled by a DDP-416 with 4K of core. This machine passes characters to and from the terminals, provides echo-back for transmission verification, provides some buffering, and passes characters one at a time into the control computer. The control computer and the job computer both share moving head disc files for data interchange, and they pass control information thru an ICCU. The control computer is essentially the executive of the system and provides the normal interaction between the user and his programs and data files which are held on the discs. The user may build up programs and data files upon the discs and when he requests that these programs be run, the control computer will queue his request for execution on the job computer. When the job computer is ready to execute the program, it will read the necessary files from the disc and will bring in any required system programs from the system disc. It will then compute for a predetermined period of time, in the region of ¼ second, and if the job has not been completed, swap it out on to the system disc and bring in the next job for a similar period of time.

In this system, the tasks have been divided fairly cleanly between the 3 computers. There is considerable difference in the computing power of the machines, each machine being matched reasonably well to the task required of it. The communications computer is a DDP-416 with 4K of memory, while the control

computer and the job computer are both DDP-516s with 32K of memory. The use of the multiple computer configuration has considerably increased the power of this system over that which would be possible with all functions undertaken in one machine, and it has simplified the task of implementing the software and of adding modifications.

As a simple example of the independent usage of the computers, the control computer and the job computer can be isolated from the normal time sharing function and used for software development by the system programmers, while the front end communications computer can remain online to the terminal users. Clearly the terminal users cannot do their normal computation, but when they attempt to sign on the communications computer can reply to them with a standard message informing them of the state of the system and when it will be back on the air. This is much more satisfactory than receiving no reply at all to an attempted sign-on and having to make a telephone call to verify the state and availability of the system.

The two applications so far discussed are indicative of how mini-computers can take on tasks normally assigned to larger machines and provide benefits of low cost, separation of programming tasks, and economical provision of redundancy. In examining these and other similar applications of small computers, various questions arise which need answering if these mini-computers are to be as widely used as I believe they should. The main points of discussion are:

1. *Shared peripherals versus shared core memories.* The two applications mentioned have used shared discs as a means for transferring data between computers, but an alternative is to share core storage or at least portions of it between two or more processors. This has the immediate advantage that data is accessible to both machines without any need for an ICCU, and no time is lost in making data transfers. There may, however, be hardware difficulties in that shared core may cause a slowing down of the normal operation and can negate the conceptual speed of advantage. It also becomes more difficult to provide redundant backup for the shared core memory, and the attempt to do so almost inevitably causes further slowing down of the effective cycle time. Probably the optimum solution is the large private memory on each processor, a small somewhat slower shared memory with redundant backup, to provide data transfer between machines, and a large high speed shared disc to provide the basic bulk data transfer.

2. *Efficient, simple, modular operating systems are required.* If multiple mini-computers are really destined to effectively challenge the large computer business in the real time application area, it is vital that they develop some sophistication and maturity in their software systems. It is impractical to develop new software for every new application and an attempt to do so will simply delay the introduction of these machines. Large, all embracing operating systems are not required, but simple, high speed, standard executives are badly needed. These should provide high speed handling of interrupts, simple task dispatching, clearly defined interfaces with application programs, and the ability for the user to start very simply and elaborately by adding modules as required to do his particular job.

3. *Large data storage peripherals.* Most mini-computers have only been available with relatively small bulk storage devices, because their designers anticipated that these machines would only be used on small scale applications. The realization that large scale applications are also appropriate for mini-computers means that large bulk storage devices, 10 million to 100 million or more characters, are required. Because of the inherently limited core storage capabilities of mini-computers, these bulk store devices must have fast access. Moving head discs are the only acceptable devices at present but even these are too slow in many applications. A typical moving head disc provides approximately 10 random accesses per second on an average, but some applications require that this be increased by an order of magnitude. Fixed head discs can be used but it may also be possible to design some form of hardware queueing into the disc file controller. This would enable the computer to output a series of requests, perhaps 10 or 20, to the disc file controller, which would then compare its current position simultaneously with all the stored access requests. It would then automatically make those transfers which were closest to its current position and so minimize the average disc latency. Assuming requests for access are made at random positions on the disk, the effective reduction in latency would be dependent on the number of requests that could be stored in the controller and searched simultaneously. It seems plausible that an improvement of 5 or 10 times could be made.

4. *Some problems will always require large machines.* When a problem requires extensive scientific calculation, long word length, hardware floating point, and extensive demands on core storage, it will always require the use of large machines. This will also be true of problems that cannot reasonably be broken down into small component parts. In these cases mini-computers could still be expected to serve as peripheral processors around the large machine.

5. *Manufacturer support.* The widespread use of mini-computers will depend on the support available to

design and implement actual systems since ideas for applications will always exceed the supply of those capable of seeing the application through to successful implementation. It will be essential for manufacturers to supply simple and efficient software modules, all useable with each other, and supported with high quality documentation. Also implementation assistance, effective field service, and maintenance support will be needed. System analysis and design may also come from the manufacturers, in some cases, but will be more likely to be provided by consultants. Manufacturers must provide meaningful training courses and effective manuals on the equipment, the software, and its potential applications. Too few people know what to do and those who do must generalize, write it down, and distribute it as widely as possible so that others may learn.

Mini-computers have a great future limited more by our collective ability to understand how they can be used than by an deficiences or omissions in the hardware.

Teleprocessing systems software for a large corporate information system

by HO-NIEN LIU and DOUGLAS W. HOLMES

Pacific Gas and Electric Company
San Francisco, California

INTRODUCTION

One of the functions of management is to control the organization in such a way that it responds to changes and deviations in the optimum manner.

The magnitude of the deviation from the established goal often depends upon the length of the delay in response, any deviation from the best performance objectives must be quickly detected and corrective measures applied promptly.

A fast response corporate information system is designed to accommodate this criterion with the following capabilities:

1. *Keeping the Corporate Data Base Freshly Updated*

   Source data may be transmitted directly into the computer to improve the efficiency of the information flow, thus providing prompt and accurate collection of data from widely dispersed areas. This capability can at least provide the following benefits:

   - Reduction in human waiting time.
   - Reduction in idle resources.

2. *Extending the Usage of the Corporate Data Base*

   New applications could be added to provide benefits not previously available.

   - Direct exchange of information with the corporate data base helps users in diverse locations keep abreast of rapidly changing events. For example:
     - Immediate presentation of operating status aids decision making.
     - Rapid transmission of decisions to the point of execution can be accomplished.
     - Swift distribution of decisions to the associated parties for supplemental decision-making are completed within the time frame.
     - Timely feedback of the results of the decisions allows adjustments to the operating environment in an incremental manner.

A well planned and developed teleprocessing system will provide the backbone of a fast response corporate information system. The remainder of this paper describes the requirements, strategy, facilities, and actual implementation of such teleprocessing system.

SYSTEM REQUIREMENTS

The following requirements are essential for a teleprocessing support of a growth-oriented corporate information system.

1. *Support for a Variety of Terminal Types*

   Each terminal installation must be reviewed to determine the specific terminal type which can best handle the types and volumes of information processing typical of that location. The system must be capable of supporting, in addition to the standard devices, several special devices tailored to satisfy special situations.

   - The standard devices will include:
     - Typewriter terminals
     - CRT terminals
     - Low-price, short-message terminals for data entry
     - Card readers, card punches, and line printers for remote locations.
   - The special devices could include:
     - Analog transducers
     - Process control computers

2. *Centralized Control of Tele-Communications Network*

   To assure efficient information flow and optimal utilization of the communications network, control of the teleprocessing system should be centralized so that resources can be allocated dynamically to satisfy changing demands. Conventional systems

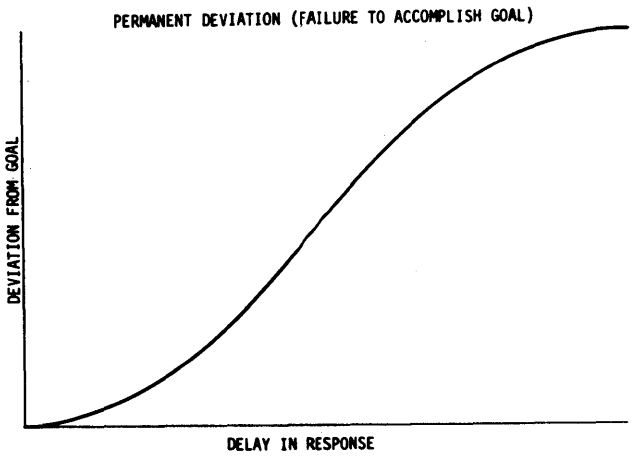


Figure 1

have often allowed segmentation of the system resources into disassociated subsystems between which temporarily unused resources cannot be shared.

For a large scale party-line (multidrop) network, provisions should be made to maintain a network discipline that will ensure increased system efficiency as well as dependable service. For example, continuing to poll a malfunctioning terminal which fails to reply degrades service to other terminals on the same line and wastes central processor time. Such a terminal, therefore, should be deleted logically from the network until it is capable of replying.

By contrast, when maintenance personnel are testing a malfunctioning terminal on-line, the system must poll this terminal as usual until testing is completed.

The network control program should provide at least the following functions:

- Polling and addressing network discipline.
- Threshold error counters to allow automatic deletion of malfunctioning lines and terminals.
- Diagnostic terminal mode which bypasses the automatic deletion of malfunctioning lines or terminals to allow for on-line hardware maintenance.
- Manual stop and start for lines and terminals to allow console operator to control network.

3. *Support for a Variety of Message Types*

Each application project should be able to select the message types best suited for its needs. The system should support a variety of basic message types which may be used independently or as building blocks for more complex activities.

These basic types of messages are:

- INQUIRY: An operator may ask predefined questions by specific transaction codes.
- RETRIEVAL: A user may select and examine information elements from the data base.
- DATA ENTRY: An operator enters new information into the data base, whether update occurs immediately or later.
- JOURNAL: An application project reports the status of transactions previously processed.
- MULTIPLE DESTINATION: A message processing program responds to a single request with messages directed to two or more locations.

A complex activity example:

- DATA CHANGE (INQUIRY + DATA ENTRY): An operator inquires into the data base and then enters changes based on the inquiry response.

4. *Versatile and Balanced Message Control Facility*

In support of these message types, the message control program should also provide the following services:

- HEADER BUILDING: Identification, time stamping, routing, and classification of messages to permit off-line analysis of message flow in addition to on-line control.
- QUEUE MANAGEMENT: For a large real-time system, the interval between message arrivals is often less than the service time so that messages cannot be processed serially nor can the system keep up with the demand for its resources. The resulting backlog of messages must be managed to smooth out peak loads and provide a tolerable response time.
- PRIORITY MANAGEMENT: Certain activities are of such importance that they demand immediate attention regardless of the backlog of other messages. A priority scheduling mechanism would permit such activity to avoid long waits in queues by providing express routes throughout the system.

5. *Efficient and Easy Applications Programming*

Economic considerations require an approach be taken which reduces programming, testing, and maintenance costs of message processing programs. The teleprocessing system should present an interface which permits such cost reduction.

6. *Testing Provision for Message Processing Programs*

To facilitate testing new or modified message processing programs in an actual operational environment without endangering the on-going operations. The system should protect at least the following resources:

- DATA BASE: Retrieval of data elements from the data base should function normally for the testing program, but any attempts to update the data base, directly or indirectly, must be intercepted.
- OPERATIONAL PROGRAM: A different storage protection key should be assigned to the testing programs.

The system should also provide the following functions:

- MESSAGE TRACE: Print or log every work area associated with a testing program to show the message in different stages during processing as a diagnostic aid. This function should be available by request for operational programs also.
- REFRESHING: After trying a specific condition to which the testing program fails to respond normally, it is desirable to refresh the copy of the testing program from the library so that different conditions can be tested to speed up the debugging cycle.
- TASK INDEPENDENCE: If one testing program fails, the system must take action for abnormal termination of this individual program (subtask); however, all the other programs in the same region should continue processing independently of this failure.
- TIME LIMITING: A program should be terminated if it does not complete within a specified time limit. This function is of value for operational programs but especially for testing programs to break tight programming loops.

7. *Data Base Security*

To protect the integrity of the corporate information system data base, security measures must be provided against unauthorized update and retrieval of privileged information.

Security should be a function of the operator's level of authority, the location of the terminal, the transaction code, as compared to the sensitivity of the data element.

8. *System Reliability*

A real-time information processing system must demonstrate its reliability to its users.

There are three aspects to reliability in any system:

- ENDURANCE: Protection against failure of its own programs, and graceful degradation of the system under adverse conditions.
- RECOVERY: Provision for restarting the system close to the point of failure after the disturbance has been removed or corrected.
- AUDIT TRAIL: Each day's message log should be retained for a period of time to permit reconstruction of a single event or a sequence of events which led to failure of a program module or the system.

After the fact analysis is often the only technique possible for problem identification/solution in a real-time environment. Some provision should be included for a computer search of the message log when specific selection criteria permit.

9. *Facility to Evaluate System Performance*

Usage statistics should be gathered to detect problem areas of the system worthy of special attention, so that solutions can be implemented to improve:

- Main frame through-put
- Network traffic
- Terminal operation efficiency
- Application program proficiency

STRATEGY OF THE SYSTEM

The principal strategy entails the reduction of redundant coding otherwise inherent in the massive application programming effort by shared system modules, wherein the following disciplines should be imposed on the system directly or indirectly:

- Application Program Proficiency
- Network Traffic Efficiency
- Terminal Operating Efficiency
- Main-Frame Throughput Efficiency

Let us define the term "application program" as referring to a message processing program tailored to handle one or more varieties of messages as identified by the transaction codes.

The three stages of application program structure described below will demonstrate the progression of teleprocessing software architecture for a large corporate information system.

STAGE 1: *Centralized Data Management Functions for All Application Programs*

A previous paper (1) has described in detail how to centralize the data management functions to obtain

MESSAGE PROCESSING FUNCTIONS EXCLUDING

FILE MANAGEMENT FUNCTIONS

1	INPUT BUFFER	MESSAGE IN TERMINAL FORM
2	INPUT EDITOR	MESSAGE NORMALIZED IN FIXED FORM
3	DATA VALIDATION	VERIFY INPUT DATA
4	PROCESS ROUTINE	PROCESSING LOGIC
	FILE ACCESS	EXCHANGE INFORMATION FROM DATA BASE
5	OUTPUT EDITOR	MESSAGE FORMATED TO TERMINAL FORM
6	OUTPUT BUFFER	MESSAGE IN TERMINAL FORM

Figure 2—Stage 1

the following benefits:

- Reduction of Core Memory Requirement
- Reduction of Program Loading Time
- Centralized Control of Shared Data Base
- Optimal Allocation of Resources Associated with Shared Data Base
- Flexibility in File Design and Record Layout

After excluding the data management function from an application program (Figure 2), the following six key functions remained to be performed:

1. *Input Buffer*

   Get message from message queue as it arrived at input buffer; the message string is in original terminal form, containing terminal control characters and varies in length and format.

2. *Input Edit*

   Normalize the input message string to fixed form by interpreting the terminal control characters, replacing absent characters and fields with nulls. Because different types of terminals have unique sets of control characters and logic, an application program that contains this function will always be dependent on the type of terminal and its logic.

3. *Data Validation*

   Validity check the information content of the incoming message string to intercept bad data and send out error messages so the user may correct and reenter the message.

4. *Process and Access Data Base*

   Process message content and exchange information with the corporate data base. Construct the logical response message to the user in fixed form without terminal control characters.

5. *Output Editor*

   Convert the logical response message from internal format to display format inserting terminal control characters for transmission. If there is more than one message to be sent to different types of terminals, construct different message strings to corresponding terminals.

6. *Output Buffer*

   Dispatch message in terminal form.

STAGE 2: *Independence of Application Program From Terminal Hardware Characteristics*

Shared message editors normalize input messages and format output messages in order to isolate the application programs from the tedious function of terminal control character interpretation.

Several advantages are derived from this approach:

1. *Programming Proficiency*

   - One application program can handle similar information from several types of terminals each with a format most suited to its special features.
   - Shared message editors permit optimization of terminal characteristics at low programming cost since they need be programmed only once.
   - New terminal types may be added and input/output display formats redesigned without application reprogramming.
   - High-level languages, such as COBOL and PL/1, can be easily applied to process fixed format message records.

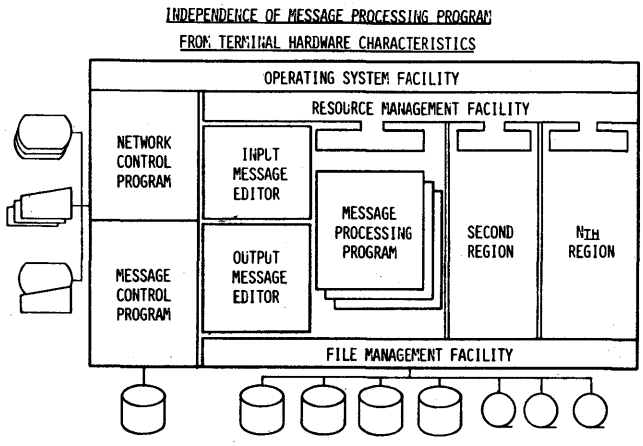


Figure 3—Stage 2

- Applications programs are easier to design, program, and test.

2. *Main-Frame Efficiency*
   - Static core requirements for application programs and work areas are reduced.
   - Application programs process a message more quickly, reducing the dynamic core requirements, measured in bytes occupied per second.

3. *Network Efficiency*

   Optimized use of terminal control characters shortens the message length, conserves message transmission time, reduces line load, and permits an increase in the number of terminals per line; the communications network may then comprise fewer lines at a sizable reduction in installation and maintenance costs for a given number of terminals.

4. *Operator Efficiency*

   Optimal use of terminal format control characters increases operator efficiency as much as it relates to display readability and input cursor control. Since the application program is truly independent of the display format, it need not be changed when a display format is redesigned or modified. This feature simplifies making improvements in terminal display design formats.

STAGE 3: *A Single Retrieval Module Replaces Many Application Programs*

Progressive reduction of redundant coding from Stage 1 and Stage 2 application programs have already placed the following functions in the shared system modules.

- Network and Message Control
- File Definition and File Access

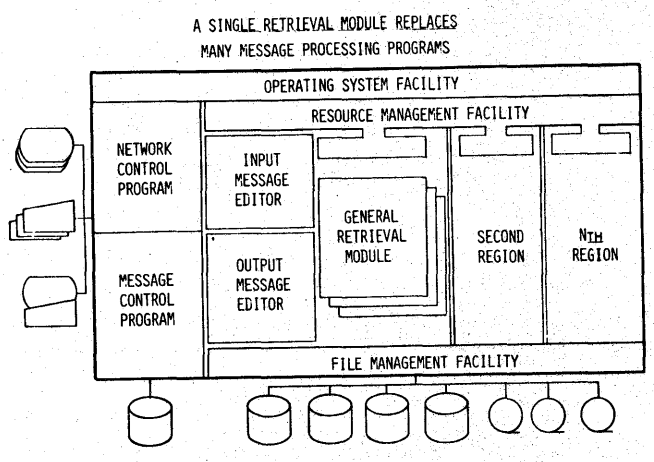


Figure 4—Stage 3

- Data Definition and Data Retrieval
- Input and Output Buffer Management
- Input Message Normalization and Editing
- Output Message Formatting

Only three functions remain to be performed by the application program.

- Input Data Validation
- Processing logic and the interface with file management programs for data retrieval from the shared data base
- Pattern editing for output message; i.e., insert decimal point, comma, $, etc.

Expanding the input message editor to perform the function of "input data validation" and the output message editor, "pattern editing," there remains only one function for the message processing program, and even this very last function can be performed by shared system modules.

- A shared system module can obtain information from the message descriptor to request data element retrieval from the data base via the data management modules.
- For most basic message types, such as INQUIRY, RETRIEVAL, DATA ENTRY, JOURNAL, etc., the processing logic can be easily represented by a simple list which defines the processing path through and within the shared system modules.

Therefore, if we create a simple list for each transaction code, the shared system modules can perform the required processing logic without recourse to an application program except when extraordinary processing logic occurs. The Stage 3 teleprocessing system will add benefits in addition to those previously derived in Stage 2.

1. *Programming Proficiency*
   - Shared input data validation and output pattern editing permit optimization in program design and efficiency.
     - Input data validation can be designed, coded, and tested in optimal fashion at low programming cost since they need be programmed only once.
     - Standard error messages can be generated directly.
   - Application programming, testing, debugging for most transactions are eliminated.

2. *Main-Frame Efficiency*

   A single resident reentrant module replaces many many application programs, eliminating the roll-in, roll-out time.

3. *Network Efficiency*

A well designed input data validation and error message notification technique can effectively cut down the amount of bad message traffic in the network.

4. *Operator Efficiency*

Standard error messages make it easy for the operator to take corrective action on bad input data; the problem of inferring the same meaning from different error messages coded by programmers in various application programs is avoided.

SYSTEM LOGIC FLOW

The system support package comprises the regional resource manager, the two message editors, the retrieval module, and the data control manager. Refer to Figure 5. The following paragraphs describe the general system logic flow.

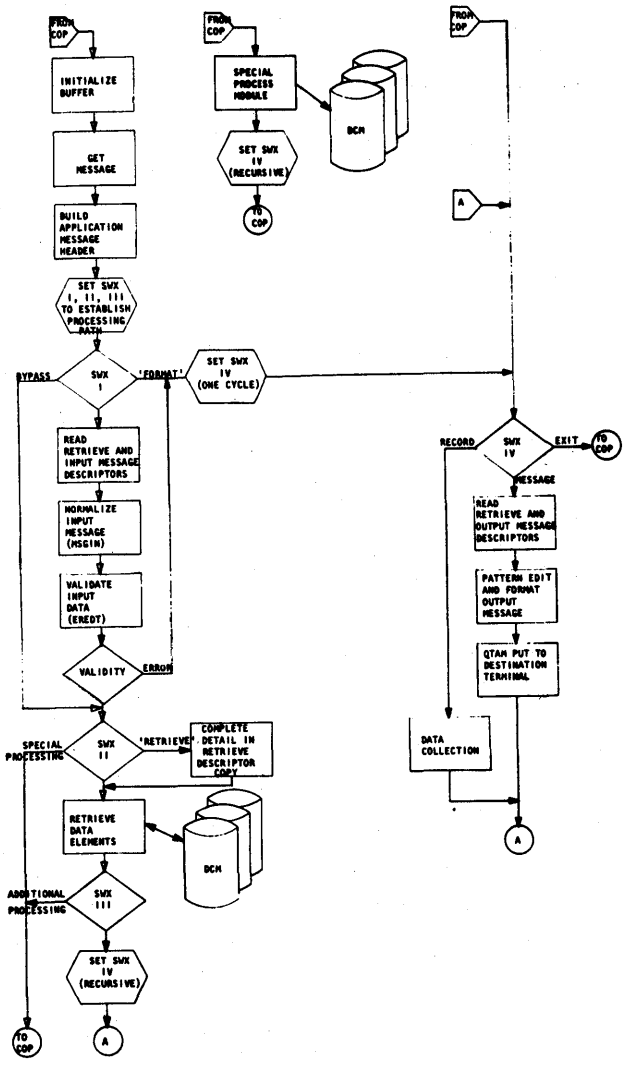


Figure 5—System logic flow

1. *Initialization*

At the beginning of the day, the regional resource manager performs the following functions:

- Builds program and descriptor directories to expedite the paging activities.
- Opens the message queues and on-line files.
- Prepares the shared buffer pools.
- Loads the resident modules.
- Attaches input and output message editors as operationally independent subtasks.

When initialization is complete, the input editor is activated and begins to scan the input queues for a message to process.

2. *GET Message*

Scanning of the message is accomplished by means of a table of displacements into a list of message queue control blocks. A pointer to a control block may appear up to four times depending upon the priority assigned to the message queue.

If a scan of the entire list fails to find a message, scanning is suspended for a predetermined period of time and then recommenced.

3. *Application Message Header*

After a message is gotten from the input queues, the input message editor constructs a message header for the application program (see Appendix II). This header serves the following functions:

- Passes pertinent information relevant to:
  - Message disposition.
  - Message status.
  - System status.
- Passes information to the application program in a form readily usable by high level languages.
- Permits the application program to pass certain information back to the system.

4. *Process Routing*

The input editor obtains the process routing information from the transaction code table and constructs a control list for the regional resource manager.

For the majority of transactions, this control list will make no reference to an application program. For those special transaction codes requiring complex processing logic, however, a reference to the

associated application program will be included in the list.

5. *Message Format*

The message may simply be a request for a message format. The input message editor posts control to the output message editor to generate a message format (captions and control characters) based on the information in the message descriptor.

6. *Normalize Input Message*

The input message editor processes the elements in an order controlled by the input message descriptor (Appendix I). The input message descriptor sequentially defines the attributes of each message field:

- The starting position in vertical and horizontal coordinates.
- The maximum length.
- A field designated as caption will be eliminated from processing.
- A field designated as mandatory information must be present or the entire message will be rejected.
- The retrieval descriptor index points into the retrieval descriptor so that additional editing may be performed.

The retrieval descriptor defines additional attributes for the message field:

- The length and location of the area reserved for the message field.
- The data class expected; i.e., numeric or alphameric.

Based on the above information, the input message editor performs the following functions:

- Checks for invalid characters, such as a letter in a numeric field, posts error condition if invalid character found.
- Deletes punctuation, such as commas in a numeric field.
- Aligns to the left or right.
- Truncates if the input message descriptor length exceeds the retrieval descriptor length, such as if the operator included too many decimal positions.

7. *Data Validation*

The retrieval descriptor serves for both editing and data validation. As many as 256 data validation routines may be programmed to permit the choice of an appropriate validation technique. Some examples of checking.

- RANGE: Test the numerical value of a data field against a predetermined range of values.
- CODE: If the input data field is a code argument in a table, the system will perform a table look-up to determine if it is a valid argument.
- RECORD KEY: If the input data field references a record key in an on-line file, the system can issue a read (key) against that file to determine if it is valid.
- FIELD ASSOCIATION: When one or more input fields depend upon the value of another input field, the system can match them against a predefined associative decision table.
- DATE: The following tests can be applied to a date field:
  - Any valid date.
  - A holiday.
  - A work day.
  - Today's date.
  - Test a range of predefined work days from today's date.
  - Test a range of predefined elapsed days from today's date.

8. *Standard Error Message*

If any errors have been identified, a standard error message is prepared.

The following considerations are taken to design the standard error messages:

- FIXED LOCATION: Error messages always appear in the same location to attract the terminal operator's attention. For example, all the error messages will appear on the last two lines of the CRT screens.
- MULTIPLE ERRORS: To conserve network efficiency and eliminate unnecessary traffic arising from bad messages, the system will handle up to four errors per message at a time.
- STANDARD PHRASE: The error message will reference the specific input field and indicate the kind of error the system detected for that field.

9. *General Retrieval Module*

The message may have been a general retrieval request. The retrieval descriptor would then have been a core-resident skeleton sufficient to satisfy the normalization and validation routines. The data identification information (element control numbers) supplied by the operator will be inserted in a copy of the skeleton so that data retrieval may proceed.

The retrieval routine builds a list containing the file name, the record key, and one or more data

element control number and receiving area pairs, and requests the services of the data control manager. Nulls are returned for a data element when the operator's or terminal's security clearance is less than that assigned to the element. A status code informs the retrieval routine of any abnormality. The retrieval descriptor defines whether an abnormal status code is to be ignored or considered an error.

10. *Output Message Editor*

The output message editor pattern edits the data field if it is required, prefaces whatever control characters are necessary to position the message character string in the terminal format. It has the following three modes of operation:

- Device With Non-Formatted Memory
  - Blanks between graphics in the same line and blanks at the front of the line are replaced everywhere possible with control characters.
  - Lines are truncated on the right after the last graphic.
- Device With Formatted Memory: When the transaction requires a new format at the device.
  - The memory of the device is cleared.
  - Fields marked as variable in the output message descriptor (device dependent option) are inserted in the message string full size without the blanks suppressed, even if they are completely blank.
  - Blanks in caption and format fields are replaced with control characters wherever possible.
- Device With Pre-Formatted Memory: When the transaction requires the same format as is at the device.
  - Caption and format fields are omitted.
  - Blanks in variable fields are replaced with control characters wherever possible.
  - The effect of each control character is considered with respect to the existing format.

11. *PUT Message*

This routine places the prepared response message in the proper destination queue for dispatch to the receiving terminal.

12. *Termination*

A privileged transaction code allows the console operator to terminate the teleprocessing system.

The system termination module directs the message control region to discontinue polling on all lines as soon as incoming messages have been received.

When polling has been discontinued, the input editors are directed to return control to the regional resource manager whenever they find the process queues empty. Normally, they wait a predetermined period of time and then scan the queues again.

When all message processing is complete, the regional resource manager terminates the teleprocessing system.

SYSTEM FACILITIES

This section will briefly describe some system support functions not mentioned in the previous section:

1. *Terminal Start-Up Procedure*

An operator must log-on before attempting any other activity on a terminal. For his own protection, he should log-off when his work is completed or during an interruption in which he leaves the terminal.

When log-on occurs, an employee number is entered in the terminal table. This employee number is used to set up individual restrictions on the terminal and to facilitate error and security violation tracing. Each time a log-off is processed, a corresponding log-on is required before business can be resumed.

A second operator may log-on at a terminal without the previous operator logging off; the second operator's employee number and restriction code replace the first's.

2. *Data Base Security*

Six modes of operation are supported for terminals:

- Business: Normal mode for business work. Most applications functions are valid in this mode. INQUIRY, RETRIEVAL, DATA ENTRY, DATA CHANGE, MULTIPLE DESTINATION and JOURNAL are available and work as defined.
- Training: Operator training mode for practicing business work.
  - INQUIRY and RETRIEVAL work as defined.
  - DATA ENTRY and DATA CHANGE appear to the operator as defined but fail to update the data base.
- Supervisor: Extended mode for business work. All applications functions are valid in this mode. At the application's discretion certain transactions may be reserved for supervisor mode or more information may be passed in this mode.

A terminal in supervisor mode may:

- Put another terminal in the same office in supervisor mode if that terminal is authorized for supervisor mode.
- Display the employee presently logged on a particular terminal.
- Copy a message to another terminal in the same office.

- Diagnostic: Systems aid for on-line engineering maintenance; message directed to special diagnostic programs which display generated status information on the console terminal.
  - A terminal in diagnostic mode may return itself to any mode authorized for it.
  - It may copy any terminal in training mode and any terminal may copy it.
- Console: Network control terminal located at the computer console; terminal, line, and transaction code status tables may be altered from this terminal.
- Master: Network monitor terminal; permits dynamic observation of system for debugging and audit control. Master mode terminals may change the mode status of other terminals to any mode of operation, including master.

3. *Network Monitoring*

Hardware errors will be analyzed and may cause the following actions to be taken:

- Send an Error Message
  - Describes the error to the console operator.
  - Describes action being taken by the system.
  - Suggests action which should be followed by the operator.
- Manually start or stop a terminal, line segment, line, or group of lines from operation on request.
- Automatically stop a terminal, line segment, or group of lines, depending on error which occurred.
- Redirect messages to an alternate terminal (which may be a different type) when the original destination terminal has any type of hardware error which renders it unable to transact business.

4. *Formats*

For data entry, a formatting facility preformats a terminal's memory with caption material and control characters at the operator's request. The operator indicates the particular transaction format required by suffixing a letter F to the associated transaction code. The facility responds with the format, and the operator simply fills in the data.

If the information content of the message is acceptable for processing, the system restores the variable fields to blanks, and the operator can proceed to the next transaction if it is the same. If it is not, a new format may be requested.

5. *Data Collection*

A data collection facility stores audit trail records created by application modules which update the data base and transaction records destined for off-line batch processing programs. These records are sorted and cataloged by off-line programs for easy retrieval at the end of the on-line day.

6. *Testing Program Library*

A special program module library contains all test status programs. Any module loaded from this library is automatically placed in test status to protect the integrity of the data base from unproved routines.

IMPLEMENTATION OF THE SYSTEM

The teleprocessing system package has been written in IBM–360 Operating System Assembly Language (ALC).

1. It is fully interfaced with the IBM–360 operating system MVT (*m*ultiprogramming with *v*ariable number of *t*asks) environment.
   - The application programs and the system programs operate as independent subtasks of the regional resource manager; abnormal termination of a subtask will not stop the remaining subtasks in the region.
   - The package is not tied to any particular release of O/S; hence, if a new version is released, there should be little effect on this package.
2. The teleprocessing system package takes full advantage of the existing operating system facilities.
3. It is intended to interface with all the operating system supported languages (COBOL and ALC interface have been implemented).
4. The entire package has been designed to be dynamic in nature; that is, all programs are load modules. They are not linkage edited into the application program; thus, the package may be redesigned and improved without any appreciable effect on the application programs.
5. The entire package has been programmed in reentrant code.
6. The system has been coded in a modular fashion. Each routine was individually coded, tested in detail, subgrouped, and finally all routines were combined together.
7. The message control and message processing regions are independent of each other to permit relocation

of the control program to a front-end communications computer when the network size warrants the change.

8. The hardware anticipated over the next several years includes two large central processors with a million bytes of main memory, supported by smaller satellite computers and a score of multi-drive disk storage units. The system is being designed to support several hundred terminals, most of which are expected to be high speed CRT display units.

ACKNOWLEDGMENT

The authors wish to thank Mr. J. R. Kleespies for his encouragement and support; Mrs. G. L. Kenny, Messrs. R. A. DaCosta and P. A. Terry for their dedicated efforts in design and programming; Mrs. L. J. Fiore for her careful typing; Mr. R. P. Kovach, University of California, for his early research and design of the system; and Messrs. F. J. Thomason and J. W. Nixon of Haskins & Sells for their invaluable advice.

BIBLIOGRAPHY

1 H LILU
A file management system for a large corporate information system data bank
FJCC Proceedings 1968

2 J MARTIN
Design of real-time computer systems
Prentice Hall 1967

3 J MARTIN
Programming real-time computer systems
Prentice Hall 1965

4 *Advances in fast response systems; fast response system design and auditing fast response systems*
EDP Analyzer February 1967—March 1967 and June 1967

5 C HAUTH
Turnaround time for messages of differing priorities
IBM Systems Journal Volume 7 No 2 1968

6 W P MARGOPOULOS others
Teleprocessing system design
IBM Systems Journal Vol 5 No 3 1966

7 M G JINGBERG
Notes on testing real-time system programs
IBM Systems Journal Vol 4 No 1 1965

8 J D ARON
Real-time systems in perspective
IBM Systems Journal Vol 6 No 1 1967

9 J L EISENBIES
Conventions for digital data communication link design
IBM Systems Journal Vol 6 No 4 1967

10 L F WERNER
Software for terminal-oriented systems
Datamation June 1968

11 J DIEBOLD
Thinking ahead—Bad decisions on computer use
Harvard Business Review January-February 1969

12 *Heading format for data transmission*
A USAAI Tutorial Communications of the ACM

APPENDIX I

Input message descriptor (fixed form)

	BITS
1. Retrieval Descriptor Index	8
2. Message Field Length	7
3. Mandatory Field Indicator	1
4. Line Number (Vertical Spacing)	6
5. Spare	1
6. Spare	1
7. Position (Horizontal Spacing)	7
8. Caption Delete	1
	4 BYTES

Input message descriptor (free form)

1. Retrieval Descriptor Index	8
2. Message Field Length	7
3. Spare	1
	2 BYTES

Output message descriptor

1. Retrieval Descriptor Index	8
2. Device Dependent Options	8
3. Line Number (Vertical Spacing)	6
4. Data Scan Override	1
5. Format	1
6. Position (Horizontal Spacing)	7
7. Caption Field	1
	4 BYTES

BITS

3 1. Data Class

a. Arithmetic (Right Numeric, Left Zero Fill, Decimal Alignment (0–7))

0 0 0 Binary, Display As Decimal
0 0 1 Binary, Display As Hex
0 1 0 Packed
0 1 1 Zoned

b. Alphameric (Left Alignment, Right Blank Fill)

1 0 0 Graphics (Terminal's Entire Character Set)
1 0 1 Alphabetic
1 1 0 Alphanumeric
1 1 1 Numeric

3 2. Decimal Alignment (For Arithmetic Class Only)

0 to 7 Places or Date Verification Decision Table (Replaces Decimal Alignment Table)

0 0 0 Any Valid Date
0 0 1 Today's Date
0 1 0 Any Holiday
0 1 1 Any Working Day
1 0 0 Prior Date
1 0 1 Prior Date or Today
1 1 0 Future Date
1 1 1 Future Date or Today

10 3. File ID Table Index/Code Table Number

3 4. Verification/Retrieval

0 0 0 Bypass Verification/Retrieval
0 0 1 Date Verification
0 1 0 Data Verification
0 1 1 Duplicate (Descriptor Points to Argument
Cross Index Points to Receiving Area Descriptor)
1 0 0 Verify Code Argument (Descriptor Points to Argument)
1 0 1 Verify File Key
1 1 0 Retrieve Code Function (Descriptor Points to Argument)
1 1 1 Retrieve File Element (Cross Index Points to Receiving Area Descriptor)

1 5. Pattern Edit Output Field
12 6. Displacement
7 7. Field Length
1 8. Spare
8 9. Retrieval Descriptor Cross Index (See Item 4) or Data Verification Range Table Index
16 10. DCM File Element Control Number (Binary Half Word) (See Item 4) or Associative Decision Table Index and Data Verification Routine Index

Retrieval descriptor (normalize input) (format output)

BITS

3 1. Data Class
3 2. Decimal Alignment (Ignore If Date Indicated)
10 3. —
3 4. Date Indicator (Output: Format YYMMDD as MM–DD–YY)
1 5. Pattern Edit Output Field (Input: Normalize MM–DD–YY as YYMMDD)
12 6. Displacement
7 7. Field Length
1 8. —
8 9. —
16 10. —

Retrieval descriptor (validation)

BITS

3 1. Data Class
3 2. Date Verification Decision Table
10 3. —
3 4. Verification Indicators
1 5. —
12 6. Displacement
7 7. Field Length
1 8. —
8 9. Data Verification Range Table Index
16 10. Associative Decision Table Index and Data Verification Routine Index

APPENDIX II

Application message header

TPCMSHDR The 40 byte application message header allows communication between the message editors, MSGIN and MSOUT, and the application module. Use of the information is left to the discretion of the application analyst.

TPCINTM Arrival Time of Day

A binary clock maintains the time of day in units of 1/150 second (6⅔ milliseconds). The high order byte contains binary zeros

The application may insert this field in the generated transaction records' sort keys to post by arrival sequence.

Because the conversion of this field to decimal hours, minutes, and seconds is time consuming, it is not appropriate to do so in the on-line environment.

TPCUSER User Status Flags

MSGIN initializes this field to binary zeros.

MSOUT logs it in the QTAM message header.

QDUMP retrieves it at the end of the day for application analysis

Each application may define its own coding structure. However, the codes should, in the least, describe the TPCSCODE selected and explain why, so that the application analysis can reconstruct the process condition.

TPCMODE Terminal Mode

A terminal may be placed in one of several operations modes which define how the system will react to messages from it.

1. Training—The application appears normal to the operator, but no transaction records should be generated or posted to the masterfile.

   An application may desire to maintain special files of pseudo accounts for training and testing and post these in training mode.
2. BUSINESS—The application reacts normally to all stimuli.
3. Supervisor—This mode is normally for terminal operator supervision but on occasion some business work will arrive from a terminal in 'supervisor' mode. The application may handle such work as business or may grant special privileges.

   Supervisor mode is allowed only for specific terminals and specific employees in supervisory positions.
4. Operator—This mode is normally for systems operation, but on occasion some business work will arrive from a terminal in 'operator' mode.
5. Master—This mode is normally for systems programming, but on occasion some business work will arrive from a terminal in 'master' mode.

TPCSOTRM Terminal of Origin

Each terminal has a unique five character identifier comprising:

DIVISION	1 byte alphabetic
OFFICE	1 byte alphabetic
LOCATION	1 byte numeric
UNIT	2 bytes numeric

The application may insert this field in the generated transaction records for journal distribution or as a debugging trace.

TPCDSTRM Terminal of Destination

The name redefines TPCSOTRM.

The application module may alter this field to redirect the response to a different terminal.

Such a receiving terminal must be a hard copy device.

Creation of an invalid terminal identifier will direct the response to a dead letter queue.

TPCINNR Message Sequence Number In

QTAM maintains an input message sequence number for each terminal

The application may insert this field in the generated transaction records for journal sequencing or as a debugging trace.

TPCDATE Today's Julian Date, YYDDD+.

This field is supplied for the application's convenience.

Because the conversion of this field to calendar format, e.g., YYMMDD, is time consuming, it is not appropriate to do so in the on-line environment.

TPCSCODE Transaction Code Modifier

TPCTCODE Transaction Code

A transaction code identifies an entry from the operator and the related response to the operator.

The transaction code modifier X'FO' is assigned to the entry on input and to the standard response for a valid entry on output.

The modifiers X'F1', X'F2', X'F3', X'F4' designate alternate responses selected by the application module processing the entry. They must, however, be designed to fit the display format of the standard response

X'F0' is the standard response.

X'F1' is the error description response For Data Entry and Data Change X'F1' is the standard acceptance response which re-initializes the terminal buffer and screen for the next entry

from the operator since the TP System assumes the next entry will be similar to the one just processed.

X'F2', X'F3', and for INQUIRY, X'F1' are available to the Application for alternate responses as they require.

COBOL linkage section for application message header

```
01 TPCMSHDR.
   03 TPCINTM      PICTURE S9(004)
                    COMPUTATIONAL.
   03 TPCUSER      PICTURE S9(002)
                    COMPUTATIONAL.
   03 FILLER       PICTURE X(010)
                    VALUE SPACE.
   03 TPCMODE      PICTURE X(001)
   03 TPCSOTRM.
      05 TPCTDST.
         06 TPCTDVS PICTURE X(001).
         06 FILLER  PICTURE X(001).
      05 TPCTOFC    PICTURE X(001).
      05 TPCTNUM    PICTURE 9(002).
   03 TPCDSTRM
         REDEFINES
         TPCSOTRM  PICTURE X(005).
   03 TPCINNR      PICTURE S9(002)
                    COMPUTATIONAL.
   03 TPCDATE      PICTURE S9(005)
                    COMPUTATIONAL-3.
   03 TPCSCODE     PICTURE 9(001).
   03 TPCTCODE     PICTURE X(004).
```

The selection and training of computer personnel at the Social Security Administration

by EDWARD R. COADY

Social Security Administration
Baltimore, Maryland

INTRODUCTION

How many computer systems managers have claimed their individual systems and operating environments contain a unique group of applications?

I suggest the majority answer yes. The "slow-down" in implementing third generation computer systems is implied in this answer. The dilemma that many systems managers are confronted with in the conversion from second to third generation systems is rooted in the educational process or lack of it.

This paper will present the social security data processing system, in general terms; the recruitment, selection and training systems for computer personnel and the future tasks of computers and their programmers at the Social Security Administration.

THE SOCIAL SECURITY DATA PROCESSING SYSTEM

The mission of the Social Security Administration is to operate a social insurance program for the American people. The Bureau of Data Processing and Accounts which is headquartered in Baltimore, Maryland maintains the earnings history for each person with covered earnings who is assigned a social security account number. These earnings records are kept so that when it is time to decide on a person's eligibility for benefits and on his benefit amount, his earnings history is available. To handle these tasks we have 50 computer systems and supporting peripheral gear and over 1200 personnel to program and man these systems. I would like to briefly discuss the major EDP functions to provide an overview for recruitment, selection, and training of programmers at Social Security.

The EDP activities of the Bureau of Data Processing and Accounts of the Social Security Administration can be classified into the following categories: (Figure 1)

(1) The New Account Establishment and Correction Process—This process involves the establishment of various records used to identify social security account number holders. Identifying information is maintained on printed listings by account number and on microfilm by name and date of birth. Approximately 250,000,000 names are found in this file. The establishment process also prepares the magnetic tape record to which worker's earnings information will be posted.

(2) The Earnings Record Maintenance Process—The earnings information of individuals participating in the social security program is maintained in two forms, magnetic tape for computer processing and microfilm for visual examination. Each of these earnings data files is updated four times a year.

After the earnings data is converted to magnetic tape, the individual employer reports are balanced in a computer process. Next, the balanced items are processed through a series of sorting operations which provide for the arrangement of items in social security account number sequence. Finally, the current earnings, balanced and sorted, are compared with the summary earnings tape and those records matching on account number and surname are updated. A new summary record is prepared. A microfilm record of those items which match is prepared as a by-product of this operation.

(3) The File Search—Benefit Computation—Earnings Statement Process—The magnetic tape file containing 185,000,000 summary earnings records is searched daily to obtain the necessary earnings information for benefit computation and earnings and coverage statement requests. The finder

Figure 1—EDP applications at SSA

1. NEW ACCOUNT ESTABLISHMENT
2. EARNINGS RECORD MAINTENANCE
3. BENEFIT COMPUTATION
4. REINSTATING
5. BENEFIT MAINTENANCE
6. HEALTH INSURANCE

items to be located number about 55,000 and are received from several sources. All of the requests concerning earnings information are arrayed on the finder tape which is processed through editing and sorting operations. The search of the summary earnings records is made, and the records located for claims and statement requests are written out for processing through separate operations. The desired data is prepared on appropriate forms, certified, and forwarded to the requesting individual, organization, or district office.

(4) The Reinstating Process—Each year approximately 312 million earnings items are received in the Bureau of Data Processing and Accounts for posting to individual earnings records. Of this amount nearly 3 million items are reported without an account number and are immediately deleted for correspondence. Of the 309 million items which we attempt to post slightly over 14 million reject because of an improperly reported name or account number. These rejected items are subjected to a series of computer and manual reinstating operations designed to locate and correct these reporting errors. The series of computer operations involved in these processes is based upon a thorough analyses of repetitive error statistics and the nature of the errors encountered in the reporting of account numbers.

(5) A master record of all social security beneficiaries is maintained on magnetic tape. This record, arranged in account number sequence, contains complete identification of the beneficiaries—including mailing address, entitlement data, benefit amount, and benefit payment history. The primary use of the record include adding new beneficiaries to the system, correcting and changing information already in the system, identifying beneficiaries becoming eligible for health insurance protection, updating the actual master tape record, preparing transcripts of the updated record for check printing purposes, and preparing a microfilm of the master record for visual reference purposes.

(6) The Health Insurance Identification and Enrollment Process—Monthly, the Bureau of Data Processing and Accounts searches magnetic tape records to identify those social security beneficiaries about to obtain age 65. An "Application for Enrollment in Supplementary Medical Insurance" is mailed to each beneficiary identified. A search of the summary earnings record is also made to identify non-beneficiaries about to attain age 65, and every effort is made to develop a claim for social security benefits, including Medicare. The combined identifying and response information is processed through a distribution operation to produce Health Insurance cards showing entitlement to Hospital Insurance and Supplemental Medical Insurance.

In support of these functions, the Bureau employs approximately 9,000 people. Of this number, over 1,200 persons are directly engaged in our EDP activities. In calendar year 1968, these people were responsible for the processing of over 7,000 different computer applications.

We maintain a magnetic tape library of over 160,000 reels and process on the average 5,000 reels per day. It is not uncommon to process a file of 500 or more reels in one operation, for example, in the Health Insurance operations, over 900 reels are needed each month to record information that is subsequently converted to a microfilm file.

About 82 million earnings items are posted to the 185 million master earnings accounts each quarter. The actual update operation is handled in a batch processing mode and over 250 hours of computer time are used.

An analysis of our system usage reflects the following data in terms of major application areas: (Figure 2)

(1) 33 percent to the claims operations—from the initial file of a claim through the continuing maintenance of the account for as long as a benefit is paid.

(2) 21 percent to the statistical operations—covering all phases of statistical activities.

(3) 19 percent to the health insurance operations—from the initial placement on the Master Health Insurance file through the continuing maintenance of the account.

Figure 2—Computer usage at SSA

1. CLAIMS	33%
2. STATISTICAL	21%
3. HEALTH INSURANCE	19%
4. EARNINGS	16%
5. MISCELLANEOUS	11%

(4) 16 percent to the earnings account operations—from the point of establishing the social security account through all of the postings to the master account and the policing of the account when it is in beneficiary status.
(5) 10 percent to miscellaneous functions—these include our own systems software activities, management information, and utility operations.

At the present time and during the next two to three years, we will be dedicating all the resources that we can spare from current operating demands in order to exploit the full potential of the third generation.

THE RECRUITMENT AND SELECTION OF COMPUTER PROGRAMMERS

Where do computer programmers come from? Anywhere you can find them. At SSA, we have discovered them within and without our organization; in our headquarters in Baltimore, our payment center in Birmingham and our district office in Klamath Falls, Oregon. From within the organization, they have come from a variety of occupations: correspondence clerks, secretaries, computer operators, claims examiners, etc. From without SSA, we have hired and lured a small number of experienced programmers from private industry and other government agencies. Our most lucrative area of new programmer blood has come from the selectees we have hired through the Federal Service Entrance Examination process. These trainees are generally fresh from the college campus and have developed into a cadre of valuable employees. We have been using this recruitment source since 1966. The selection system, except for the experienced hires, is based on an aptitude test score. Actually, two tests are used, one for the SSA employees and the other for the FSEE trainees.

The FSEE examination is a general abilities test which covers vocabulary, reading comprehension and quantitative reasoning. It is used by most government agencies for entry positions in a variety of career fields. The in-house test, which we call the Organization and Methods examination, was developed by SSA test psychologists and is given for three job categories: management analyst, budget analyst and computer programmer. There are three parts to the test: verbal, quantitative and abstract reasoning. Individual test items are statistically related to job success in each part. The emphasis is placed on the job relatedness of the test. Although the test may lack academic flavor it does allow SSA employees to use their backgrounds to demonstrate their abilities in the test areas. The test was developed after the jobs were studied, a validation of the test items were made and weights assigned based on the correlation of test score to job success. We have found both the FSEE and the O&M tests to be good predictors of success in the training program and on-the-job. Additionally, we have given the IBM Programmer Aptitude Test to several hundred trainees. The PAT scores, also are indicative of success in programming and correlate with the other tests. We feel the aptitude test is an integral part of the recruitment and selection system for programmers.

Figure 3—Programmer selection criteria

APPRAISALS	38%
APTITUDE TEST	35%
EDUCATION	7%
INCENTIVE AWARDS	3%
RELATED WORK EXPERIENCE	12%
RELATED OUTSIDE ACTIVITIES	5%

The mix of inputs for programming positions is also desirable because:

(1) In-house employees generally require less on-the-job training to become productive.
(2) Organizational morale is boosted when the rank and file employee makes the grade as a programmer.
(3) Selecting only in-house personnel; however, would ultimately weaken the organization, so the infusion of new blood results in strengthening the competitive spirit.

The selection for training in computer programming is based on criteria in addition to the aptitude test score. (Figure 3) For internal employees we apply numerical weights to the selection elements as follows: the employee appraisal—38%, the aptitude test score—35%, related work experience—12%, education—7%, incentive awards—3%, and related outside activities—5%.

To illustrate this point, the following is a breakdown of the employee appraisal and the respective element weights:

Element	*Outstanding*	*Above Standard*	*Meets Standard*
Productivity	7	4	2
Work Quality	7	4	2
Initiative	7	4	2
Resolution of Problems	10	6	3

Working With Others	5	2	1
Adjustments to Work Pressures	7	4	2

Total Points = 43

To give you an idea of the great interest in getting into programming work, let me mention some data from our most recent selection process.

Nine hundred and sixty-three employees filed applications in response to a bulletin board advertisement for twenty trainee programmer positions. The selection criteria was applied to each applicant. The applicants were ranked by total points. The twenty trainees were selected to attend the training program. These twenty employees had 'A' in aptitude test, most were college graduates, all had above standard or outstanding on all appraisal elements and some had several incentive awards.

Next, we subject the group to an eight week training program. It will be described in detail later. Our historical data reflects; that only thirteen of the twenty will be successful in the training program. This provides support for those employers who are extremely cautious in their programmer selection and hiring practices. We are no exception. Let me summarize the objectives of our selection system:

(a) to identify employees for a career program leading to a senior supervisory systems analyst,
(b) to provide the opportunity for in-house employees to enter this job stream, and
(c) to provide the infusion of highly qualified people from outside the organization via the Federal Service Entrance Examination.

THE COMPUTER PROGRAMMER TRAINING PROGRAM

Armed with highly qualified internal employees and FSEE candidates, as input, we give the candidates a rigorous eight week computer programming training course. The course has three phases; the first three week period consists of presentations on computer fundamentals (we use a hypothetical computer as an example), introduction to System/360 and assembly language coding for S/360. Frequent review quizzes and an examination at end of the third week are given. A comprehensive evaluation by the course instructors of each candidate is then made and unsuccessful candidates are cut from the training program. In-house employees return to their former jobs, and FSEE candidates are assigned to other responsible positions in the Administration. Approximately, one third of the class is phased out at this point.

Figure 4—Source of programmers at SSA

WITHIN	*INPUT*	*NOW*
Headquarters	572	285
Field Offices	349	141
WITHOUT		
FSEE	162	81
OTHER	10	10

The second phase of the training course consists of one week of advanced assembly language techniques and three weeks of COBOL coding. Several lectures on operating system techniques and job control language are also covered in this phase. The third phase of the course is a series of briefings by members of the programming staffs on special systems, techniques, administrative writing, operational procedures, standards, etc. (Figure 4)

Our history of using the training class as a screening device has been successful. Since 1955, we have selected 1083 employees for training, 507 are programming for us today, 359 or 33% were phased out or voluntarily withdrew during the training course. The remaining 217 have migrated into other organizations, advanced into management positions in other parts of our organization, retired or died. In analyzing, where our strength is, in terms of the most valuable long term employee, we have experienced less turnover in headquarters people than field people. Field office personnel seem to have a strain of nomad in them which shows up as soon as enough experience is gained in programming to make them marketable. The cost in attracting the field employee is high, since the costs of travel, per diem while in training, household moves, etc. are paid to lure him to the headquarters installation. For example, from 1959–1964, we brought 125 field employees into the headquarters for the training program, 46 were cut (37%); of the remaining 79 graduates only 26 (20%) are with us today in programming work.

In summing up the training program, our objectives are:

(a) to identify those who can program a digital computer, that is, to assimilate, analyze, solve problems, code solutions and evaluate results;
(b) to prepare the trainee for the on-the-job environment through training in assembly language and COBOL and the techniques for using these languages.

The training program is conducted by our own staff

of administrative specialists. The class is limited to 30 trainees. The methodology consists of a lecture-problem solving sequence which provides sufficient time for instructor-student counselling and assistance. The manufacturer's manuals are used for reference by the trainees. Several problems in each language are compiled and analyzed during the course.

Following the training course, a one year on-the-job training phase takes place. During this period the trainee is evaluated on his programming assignments. The elements used are:

(a) ability to absorb and retain information,
(b) originality and creative imagination,
(c) analytical ability,
(d) thoroughness,
(e) initiative,
(f) industry,
(g) working with others,
(h) oral expression, and
(i) written expression.

Even after our refined system of selecting programmers, we have a few trainees that cannot cope with the rigors and frustrations associated with programming work. These employees are phased into other staff or administrative positions.

The total system for selecting, training and ultimately promoting employees functions under the legal aegis of a training agreement approved by the U.S. Civil Service Commission. The salary range in this program starts at $7,639 at the entry to a maximum of $11,233. Advances at one year intervals are provided to $9,320 and then to $10,203. These raises are automatic providing satisfactory performance and meeting the time-in-grade requirements. Trainees may enter the stream at any of these levels dependent on their present salary level and experience. Beyond the $11,233 level, competitive promotional procedures are used.

THE ADP TRAINING STAFF

For those who can afford their own ADP training staff, I like to briefly mention the fruitful experience we have had with ours. In the early 1950's, the EAM days, we had the need for a variety of training courses in machine operation and wiring. One training officer was dedicated to the development and tailoring of these courses for SSA personnel.

We soon reaped benefits from this arrangement. We incorporated our own procedures in the training, conducted the courses at *our* convenience at *our* installation. This experience laid the foundation for using our own people for computer programmer and operator training in 1956. The amount of training needed initially and continuously justified the enlargement of the staff to the six instructors we have today. Last year, over 1,200 employees were trained in 78 courses of instruction. Our instructors spent over 5,000 hours in the classroom in the conduct of these courses. In addition to the initial training course, we conduct courses in FORTRAN, COBOL, OS Concepts, Job Control Language and operations courses tailored for medium and large scale systems as well as Operating Systems training for senior operators and job schedulers.

The selection of our instructor staff has been based primarily on the following criteria:

1. desire to instruct
2. high performance in the programmer training program
3. programming ability
4. strong administrative skills

THE FUTURE TASKS

The problems that we face today with the implementation of third generation systems, and will face in the future with the fourth generation, have their roots in the past. The problems of the second generation and how they were solved dictate to a large degree how we must now proceed. A look at the evolution of automatic data processing at Social Security will serve to give an appreciation of our current problems. Keeping abreast of the processing workloads is not enough, we are required to make major changes in our processes each time Congress enacts a change to the Social Security Act and often the time frame that must be adhered to is not of our choosing. Although we have many large jobs, large data files, and large volume detailed transactions, our conversion effort is not limited to the conversion of a few large jobs. Rather, our activities require the running of many jobs, both large and small. As mentioned earlier, last year we processed over 7,000 jobs: some daily, weekly, monthly, quarterly, annually, and some were one-time operations.

With second generation hardware we adhered to the concept of integrating the large computers and the peripheral systems. That is, keep the big machines going with the fastest input/output devices available and burden the smaller equipment with the necessary editing, formatting, printing, and punching. This permeated our every operation—it was a way of life. Most of the almost 500 programmers and systems analysts grew up with this concept. Each of our

programmers was imbued with a consciousness of the cost of the operation. He set about to maximize the utilization of the resources at his command, use all of the tape drives and all of the memory in the system to the extent that their use made his operation the most efficient possible. If he didn't use them, those resources would remain idle during the running of that program. At the same time, he was aware of the relatively high cost of processing a reel of tape, and therefore strove to reduce file sizes. Also, he saved tape space with special non-standard labels; by combining, where feasible, more than one data file on a tape reel; by manipulating the memory character to save space when indicators and codes were needed; and by using many sizes of variable length records. For example, in our Master Earnings file, we indicated the quarter of coverage pattern by the use of bit codes over the earnings field. I mentioned variable records, our record blocks vary from 15 characters to nearly 18,000 characters. To illustrate the effect of adding additional characters to each record in our large files, if only one character was added to each of our 185 million master earnings accounts that we search daily, would result in that file being expanded by 25 reels of tape. Since the file is processed daily, it represents substantial time and cost factors.

The preceding are some of the facts and considerations that have led us to where we are today—in the midst of converting to third generation systems. We believe that we had a most efficient second generation installation. We utilized the resources effectively and created a smoothly running program. Now, as we move forward, we have no choice but to live with, and to remain compatible with, what we created in the past, pending redesign of master records and processing systems. The biggest problem that faces us in the conversion to the third generation is the need to keep the social security program running smoothly while making a gradual transition. Each month, 25 million beneficiary checks must be mailed; new claims for OASDI benefits must be processed and added to the beneficiary reels; and the utilization of health insurance benefits must be recorded.

Returning to the training aspect, we find that a properly paced education program is the key to an orderly conversion period from second to third generation systems. To date, virtually our entire programming staff has received training on third generation systems. This training program had some problems of its own. The vetern programmers have a wealth of knowledge and experience with second generation equipment. They are skilled in their own fields. They had to start from scratch. They had to return to class; learn new concepts, and jargon hardware/software, etc.—in short, they (the pros) were trainees. They not only had to learn new programming skills, but had to keep current operations going full tilt. And all this at a time when we are working in a rapidly expanding work environment.

One of the problems we encountered in training for the third generation was the scheduling of people for these classes. The people who needed training were also needed to keep our day-to-day activities current with planned modifications, necessary changes, and scheduled commitments. We solved this by conducting half-day training sessions. Without in-house training capability, training and conversion would have been seriously hindered.

We envision that in several years a real-time claims process will be available that will permit "instant updating" of our master files. To do this we will have to eliminate our tape files and the 5,000 mountings required each day in our present system. The real-time files would take the form of large scale random access devices and mass storage devices which are capable of supporting a continuous updating process. Response to inquiry and request for action would be based on the most recent data possible which would be instantly accessible. For example, we anticipate that a district office will be able to request information over a teleprocessing system and receive a reply in the same day. The telecommunications linkage is already available—all that is needed is a means of instantly tapping the file to retrieve and forward the requested data. From 75–100 billion bytes of information will have to be accessible. Our earnings file alone will probably require 40 billion bytes. To support this vast information storage and retrieval system there will need to be high speed printers or graphic display and photocopy units in each district office. The same devices will be used in other locations where correspondence is handled or where action decisions are made outside of the automated system and inputted to the system. The same basic capabilities will enable us to process a large number of claims in the briefest imaginable time. Data will be wired by the district office, the earnings record will be summarized instantly and complete claims information will be channelled through to a point where, on receipt of a signal that no problem exists, or on receipt of correcting information, the payment of benefits will be started.

To support this system of real-time access and instantly updated input, devices will be needed which place claims application forms and reports requiring action in as direct contact as possible with the EDP system. With proper design of input documents and low equipment cost, the idea of optical scanning devices in district offices will be entertained. These optical scanners, will have to handle not only claims application data, but reports prepared by beneficiaries as well.

As mentioned earlier, the lines of communication

exist today. With the development of random access files and rapid response input/output devices, our ideal system can be attained perhaps within the next several years.

In summary, at present, Automatic Data Processing in the Social Security Administration is in a highly dynamic state of flux from second generation to third generation systems. At the same time our sights must be on the issues and problems we will face with the fourth generation. At the same time and almost in spite of this, our basic mission must remain ever dominant, the administration of the terms of the Social Security Act with all its amendments and related legislation in a timely fashion and with due regard for the rights and needs of that segment of the public whom we serve.

BIBLIOGRAPHY

1 A DRATTELL
The people problem
Business Automation pp 34–41 November 1968

2 R PASTON
Organization of in-house education
Proceedings of the 1968 International DPMA Conference pp 225–231

3 A BIAMONTE
Predicting success in programmer-training
Proceedings of the Second Annual Computer Personnel Research Conference pp 9–12

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

OFFICERS and BOARD OF DIRECTORS OF AFIPS

President

Dr. Richard I. Tanaka
California Computer Products, Inc.
305 North Muller Street
Anaheim, California 92803

Vice President

Mr. Keith W. Uncapher
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Secretary

Mr. R. G. Canning
Canning Publications, Inc.
134 Escondido Avenue
Vista, California 92083

Treasurer

Dr. Robert W. Rector
Informatics, Inc.
5430 Van Nuys Boulevard
Sherman Oaks, California 91401

Executive Director

Dr. Bruce Gilchrist
AFIPS Headquarters
210 Summit Avenue
Montvale, New Jersey 07645

Executive Secretary

Mr. H. G. Asmus
AFIPS Headquarters
210 Summit Avenue
Montvale, New Jersey 07645

ACM Directors

Dr. B. A. Galler
Computing Center
University of Michigan
Ann Arbor, Michigan 48104

Professor Anthony Ralston
State University of New York
Computing Center
4250 Ridge Lea Road
Amherst, New York

Mr. Donn B. Parker
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

IEEE Directors

Mr. L. C. Hobbs
Hobbs Associates, Inc.
P.O. Box 686
Corona del Mar, California 92625

Dr. Robert A. Kudlich
Wayland Laboratory
Raytheon Company
Boston Post Road
Wayland, Massachusetts 01778

Dr. Edward J. McCluskey
Dept. of Electrical Engineering
Stanford University
Palo Alto, California 94305

Simulation Councils Director

Mr. James E. Wolle
General Electric Company
Missile & Space Division
P.O. Box 8555
Philadelphia, Pennsylvania 19101

American Society for Information Director

Mr. Herbert Koller
ASIS
2011 Eye Street, N.W.
Washington, D.C. 20006

Association for Computation Linguistics Director

Dr. Donald E. Walker
Head, Language and Text Processing
The Mitre Corporation
Bedford, Massachusetts 91730

Special Libraries Association Director

Mr. Burton E. Lamkin
National Agricultural Library
U.S. Department of Agriculture
Beltsville, Maryland

Society for Information Display Director

Mr. William Bethke
RADC—(EME, W. Bethke)
Griffis Air Force Base
New York, New York 13440

Society for Industrial and Applied Mathematics Director

Dr. D. L. Thomsen, Jr.
IBM Corporation
Armonk, New York 10504

AFIPS Committee Chairmen

Awards
Mr. Fred Gruenberger
5000 Beckley Avenue
Woodland Hills, California 91364

Admissions
Dr. Robert W. Rector
Informatics, Inc.
5430 Van Nuys Boulevard
Sherman Oaks, California 91401

Constitution & By Laws

Mr. Richard G. Canning
Canning Publications, Inc.
134 Escondido Avenue
Vista, California 92083

Ad Hoc Conference Committee

Dr. Barry Boehm
Computer Science Department
The RAND Corporation
1700 Main Street
Santa Monica, California 90406

Education

Dr. Melvin A. Shader
CSC—Infonet
650 N. Sepulveda Blvd.
El Segundo, California 90245

Finance

Mr. Walter L. Anderson
General Kinetics, Inc.
11425 Isaac Newton Square So.
Reston, Virginia 22070

Harry Goode Memorial Award

Mr. Brian W. Pollard
Radio Corporation of America—NPL
200 Forest Street
Marlboro, Massachusetts 01752

IFIP Congress 71

Dr. Herbert Freeman
Professor of Electrical Engineering
New York University
University Heights
New York, New York 10453

International Relations

Dr. Richard I. Tanaka
California Computer Products, Inc.
305 North Muller Street
Anaheim, California 92803

JCC Conference

Dr. A. S. Hoagland
IBM Research Center
P.O. Box 218
Yorktown Heights, New York 10598

Information Systems
Miss Margaret Fox
Office of Computer Information
U.S. Department of Commerce
National Bureau of Standards
Washington, D.C.

JCC Technical Program
Dr. David R. Brown
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

JCC General Chairmen

1970 FJCC

Mr. Robert A. Sibley, Jr.
Department of Computer Science
University of Houston
Cullen Boulevard
Houston, Texas 77004

1971 SJCC

Mr. Jack Moshman
RAMSCO
6400 Goldboro Road
Bethesda, Maryland 20034

1971 FJCC

Mr. Ralph R. Wheeler
Lockheed Missiles and Space Co.
Dept. 19-31, Bldg. 151
P.O. Box 504
Sunnyvale, California 94088

1970 SJCC STEERING COMMITTEE

1970 SJCC *Steering Committee*

General Chairman

Harry L. Cooke
RCA Laboratories

Vice Chairman

William C. Tarvin
UNIVAC

Technical Program

James H. Bennett—Chairman
Applied Logic Corporation
Horace Fisher—Vice Chairman
Applied Logic Corporation

Paul Chinitz
UNIVAC

Martin Goetz
Applied Data Research

David E. Lamb
University of Delaware

Thomas H. Mott
Rutgers University

Joseph Raben
Queens College of the City of N.Y.

C. V. Srinivasan
Rutgers University

Sheldon Weinberg
Cybernetics International

Treasurer

James T. Dildine
Price Waterhouse & Co.
Stanley R. Keyser—Vice
Price Waterhouse & Co.

Secretary

Edwin L. Podsiadlo
Dataram Corp.
Ronald T. Avery—Vice
Dataram Corp.

Local Arrangements

John J. Geier—Chairman
Univac
Edward L. Hartt—Vice Chairman
N. J. Bell Telephone Co.

Public Relations

Conrad Pologe—Chairman
AT&T
J. Bradley Stroup—Vice Chairman
General Electric Co.

Special Activities

Edward A. Meagher—Chairman
Hoffman La Roche Inc.
Tom Carscadden—Vice Chairman
The Shering Corp.

Registration

Richard A. Bautz—Chairman
Axicom Systems Inc.
Mark Ricca—Vice Chairman
Comsul Ltd.

Ladies Activities

Peggy Crossan—Chairman
RCA
Rita Morley—Vice Chairman
RCA

Publications

Donald Prigge—Chairman
UNIVAC

Phillip A. Antonello—Vice Chairman
UNIVAC

Exhibits

Ed Snyder—Chairman
Lou Zimmer Organization
Herbert Richman—Vice Chairman
Data General Corp.

SCI Representative

Jess Chernak
Bell Telephone Labs

ACM Representative

A. B. Tonik
UNIVAC

IEEE Representative

N. R. Kornfield
Data Engineering Associates

SESSION CHAIRMEN, PANELISTS, DISCUSSANTS, REFEREES

SESSION CHAIRMEN

Saul Amarel
Peter Denning
Thomas De Marco
George Dodd
Robert Forest
Martin Goetz
Julien Green
Herbert Greenberg
Albin Hastbacka
Theodore Hess
H. K. Johnson
R. A. Kaenel
Alvin Kaltman
John L. Knupp, Jr.
David Lamb
James F. Leathrum
A. Metaxides
John Morrissey
Stuart L. Mathison
Marvin Paull
Howard R. Popper
Joseph Raben
James Rainey
David Ressler
Lawrence Roberts
William Rogers
Hal Sackman
William Schiesser
John Seed
R. E. Utman
Sheldon Weinberg
Kendall Wright
Karl Zinn

PANELISTS

Andrew Aylward
Roger M. Bakke
Kenneth R. Barbour
Ed Berg
H. Borko
Donald Croteau
John C. Cugini
S. H. Chasen
Steven A. Coons
Jack Dennis
O. E. Dial
Ernest Dieterich
Saul Dinman
Philip H. Dorn
S. Drescher
R. A. Dunlop
E. S. Dunn, Jr.
Joel D. Erdwinn
Alfred Ess
Robert B. Forest
John L. Gentile
Jack Goeken
A. J. Goldstein
David McGonagle
Donald N. Graham
Frank Greatorex
Kelly E. Griffith
Herbert J. Grosch
Margaret Harper
Frank E. Heart
Vico Henriques
Bertrom Herzog
Richard Hill
L. Kestenbaum
Charles Lecht
Lawrence I. Lerner
William Lewish
William R. Lonergan
Michael Maccoby
John McCarthy
Carl Machover
Sam Matsa
George H. Mealey
M. A. Melkanoff
Therber Moffett
Allen Newell
Seymour Papert
E. Parker
Mike Patterson
Ralph Pennington
Michael I. Rackman
L. John Rankin
Carl Reynolds
Leonard Rodberg
Arthur Rosenberg
Robert Rossheim
Gordon R. Sanborn
Robert Simmons
Dan Sinnott
William G. Smeltzer, Sr.
David M. Smith
William D. Stevens
Harrison Tellier
Frederick B. Thompson
Carl Vorlander
Larry H. Walker, Jr.
Philip M. Walker
William E. Ware
Frank Wesner

DISCUSSANTS

James P. Fry
A. N. Habermann
B. Huberman
Butler W. Lampson
H. W. Lawson
Gene Levy
R. McClure
R. E. Merwin
T. Kevin O'Gorman
Richard Robnett
Jerome H. Saltzer
Edgar A. Sibley
Dudley Warner
Thomas Wills-Sandford

REFEREES

Ralph Alter
Stanley Altman
Paul Baran
T. Benjamin
W. A. Beyer
Garrett Birkhoff
John Bruno
Edward Burfine
Walter Burkhard
Peter Calingaert
W. F. Chow
C. Christensen
J. W. Cooley
Peter J. Denning
Donald O. Doss
William B. Easton
R. D. Elbouin
Everett Ellin
Edward R. Estes
George A. Fedde
Wallace Feurzeig
Tudor R. Finch
James Flanagan
Franklin H. Fowler, Jr.
Margaret R. Fox
Al Frazier
C. V. Freiman
James P. Fry
Edward Fuchs
L. M. Fulton
Adolph Futterweit
Reed M. Gardner
John Gary
James B. Geyer
Clarence Giese
M. C. Gilliland
G. Golub
M. H. Gotterer
Alonzo G. Grace, Jr.
J. Greenfield
Donald W. Grissinger
George F. Grondin
W. B. Groth
Frank G. Hagin
Murray J. Haims
Carl Hammer
Fred M. Haney
A. G. Hanlon
R. Dean Hartwick
A. D. Hause
John F. Heafner
Walter A. Helbig
Bertram Herzog
Elias H. Hochman
R. W. Hockney
A. D. C. Holden
Robert L. Hooper
R. Howe
S. Hsu
Thomas A. Humphrey
Albert S. Jackson
Ronald Jeffries
R. A. Kaenel
Marvin J. Kaitz
R. Kalaba
Ted Kallner
M. S. Kepbign
Robert E. King
Edward S. Kinney
Justin Kodner
Igal Kohari
John Kopf
G. A. Korn
A. B. Kronenberg
Jerome Kurtzberg
Kenneth C. Kwan
Dominic A. Laiti
Richard I. Land
W. D. Lansdown
D. J. Lasser
Gary Leaf
Gene Levy
W. Wayne Lichtenberger
Minna Lieberman
J. Lindsay
Robert Linebarger
Dimitry A. Lukshin
R. McDowell
Charles M. Malone
Carl W. Malstrom
Paul Meissner
Leslie Mezei
Stephen W. Miller
Baker A. Mitchell, Jr.
M. Moe
Robert P. Myers
Joseph A. O'Brien
T. Kevin O'Gorman
Thomas C. O'Sullivan
Thomas Pyke
Toby Robison
G. Rybicki
Asra Sasson
Elmer Shapiro
C. K. Show
E. H. Sibley
J. R. Splear
Mary Stevens
Thomas Stockham
Fred Tonge
R. Vichnevetsky
Dudley Warner
Jerome Wiener
Calvin Wilcox
Lyle C. Wilcox
David G. Williams
Thomas Wills-Sandford
James E. Wolle
Sherrell L. Wright
Yu-chi Ho

1970 SJCC PRELIMINARY LIST OF EXHIBITORS

ACM
Addison-Wesley Publishing Company
Addmaster Corporation
Addressograph Multigraph Corporation
Advance Research, Inc.
Advanced Memory Systems, Inc.
Advanced Space Age Products, Inc.
Advanced Terminals Inc.
AFIPS Press
Airoyal Mfg. Co.
Allen-Babcock Computing, Inc.
Allied Computer Systems Inc.
Allied Computer Technology Inc.
American Data Systems
American Regitel Corporation
American Telephone & Telegraph
AMP Incorporated
Ampex Corporation
Anderson Jacobson, Inc.
Applied Data Research, Inc.
Applied Digital Data Systems, Inc.
Applied Dynamics Inc.
Applied Logic Corporation
Applied Magnetics Corporation
Applied Peripheral Systems, Inc.
Astrocom Corporation
Astrodata, Inc.
Atlantic Technology Corporation
Atron Corporation
Auerbach Info, Inc.
Auricord—Div. of Scoville Co.
Auto-Trol Corporation
Axicom Systems, Inc.
Beehive Electrotech Inc.
Bendix Corp.—Advanced Products Div.
Beta Instrument Corporation
BIT, Incorporated
Boole & Babbage, Inc.
Bridge Data Products, Inc.
Brogan Associates, Inc.
Bryant Computer Products
Bucode Inc.
Bunker-Ramo Corp.—Bus. & Ind. Div.
Business Press International, Inc.
California Computer Products, Inc.
Cambridge Memories, Inc.
Carterfone Communications Corporation
Century Data Systems, Inc.
Cincinnati Milling Machine Co.
Cipher Data Products, Inc.
Clary Datacomp Systems, Inc.
Codex Corporation
Cogar Corporation
Cognitronics Corporation
Colorado Instruments, Inc.
Comcet, Inc.
Community Computer Corporation
Compat Corporation
Compiler Systems Inc.
CompuCord, Inc.
Computek, Inc.
Computer Automation, Inc.
Computer Design Publishing
Computer Devices, Inc.
Computer Digital Systems, Inc.
Computer Displays, Inc.
Computer Learning & Systems Corporation
Computer-Link Corporation
Computer Micro-Image Systems, Inc.
Computer Operations
Computer Optics, Inc.
Computer Peripherals Corporation
Computer Products, Inc.
Computer Sciences Corporation
Computer Signal Processors, Inc.
Computer Synectics
Computer Terminal Corporation
Computer Transceiver Systems, Inc.
Computervision Corporation
Computerworld
Consolidated Computer Services Limited
Courier Terminal Systems, Inc.
Cybermation, Inc.
Daedalus Computer Products, Inc.
Dasa Corporation
Data 100 Corporation
Data Action Corporation
Data Automation Communications
Data Card Corporation
Data Computer Systems, Inc.
Data Computing, Inc.
Datacraft Corporation
Data Disc
Dataflo Business Machines Corporation
Data General Corporation
Dataline Inc.
Datamate Computer Systems, Inc.
Datamation
Datamax Corporation
Data Printer Corporation
Datapro Research
Data Processing Magazine
Data Products News
Data Products Corporation

Dataram Corporation
Data Systems News
DataTerm Inc.
Data Terminal Systems
Datatrol Corporation
Datatrol Inc.
Datran Corporation
Delta Data Systems Corporation
Digi-Data Corporation
Digital Equipment Corporation
Digital Information Devices, Inc.
Digital Scientific Corporation
Digitronics Corporation
DSI Systems, Inc.
Dynelec Systems Corporation
Eastman Kodak Co.—Bus. Sys. Mark. Div.
EG & G
EDP Technology Inc.
Edutronics
Edwin Industries Corporation
Electronic Arrays Components Div.
Electronic Arrays Systems Div.
Electronic Associates, Inc.
Electronic Information Systems, Inc.
Electronic Memories
Electronic News—Fairchild Publications
EMR Computer
Engineered Data Peripherals Corporation
Fabri-Tek Inc.
Facit-Odhner, Inc.
Factsystem Inc.
Ferroxcube Corporation
Ford Industries
Foto-Mem, Inc.
General Automation Inc.
General Computers, Inc.
General Electric Company
Gerber Scientific Instrument Co.
Gould, Inc.—Graphics Division
GRI Computer Corporation
Hayden Publishing Company, Inc.
Hazeltine Corporation
Hewlett-Packard
Hitachi
Honeywell—CCD
Honeywell—EDP
Houston Instrument
IBM Corporation
IEEE
IER Corporation
Image Systems, Inc.
Imlac Corporation
Industrial Computer Systems, Inc.
Info-Max
Inforex, Inc.
Information Control Corporation
Information Data Systems, Inc.
Information International
Information Storage Systems, Inc. (ISS)
Information Technology Inc. (ITI)
Information Technology, Incorporated
Infotec, Inc.
Infotechnics, Inc.
Infoton, Inc.
Interdata, Inc.
Interface Mechanisms, Inc.
International Computer Products, Inc.
International Computers Limited
International Data Sciences, Inc.
Interplex Corporation
Iomec, Inc.
Jacobi Systems Corporation
Kennedy Company
Keymatic Data Systems Corporation
Kybe Corporation
Litton—Automated Business Machines
Litton—Datalog Division
Lockheed Electronics Company
Logic Corporation
McGraw-Hill Book Company
3 M Company
Madatron Corporation
MAI
Mandate Systems, Inc.
Marshall Data Systems
Mechanical Enterprises, Inc.
Megadata
Memorex Corporation
Memory Technology, Inc.
Merlin Systems Corporation
Micro Switch—Div. of Honeywell
Micro Systems Inc.
Microwave Communications of America, Inc.
Milgo Electronic Corporation
Modern Data
Mohawk Data Sciences Corporation
Monitor Data Corporation
Monitor Displays
Motorola Instrumentation & Control Inc.
NCR—Industrial Products Division
NCR—Paper Sales
Nortec Computer Devices, Inc.
Nortronics Company, Inc.
Novar Corporation
Novation, Inc.
Omega-T Systems Inc.
Omnitec—A Nytronics Corp.
Path Computer Equipment
Penta Computer Associates, Inc.
Penril Data Communications, Inc.

Peripheral Dynamics Inc.
Peripheral Equipment Corporation
Peripherals General, Inc.
Perspective Systems, Inc.
Potter Instrument
Precision Instrument Company
Prentice-Hall, Inc.
Press Tech, Inc.
Princeton Electronic Products
Quantum Science Corporation
Quindar Electronics, Inc.
Raytheon Company
RCA Corp.—Electronic Components
RCA Corp.—Graphic Systems Division
RCA Corp.—Memory Products Division
RCA Ltd.—Divcon Systems
Reactive Computer Systems
Redcor Corporation
Remex Electronics—Div. Ex-Cell-O Corp.
Research & Development (F. D. Thompson)
Resistors
RFL Industries, Inc.
Rixon Electronics Inc.
Rolm Corporation
Royco Instruments, Inc.
Sanders Associates, Inc.
Sangamo
Scan-Data Corporation
Scan-Optics, Inc.
Scientific Time Sharing Corporation
Scope
The Service Bureau Corporation
Shepard/Div. of Vogue Instrument Corp.
Singer—Friden Division
Singer—Librascope
Singer—Tele-Signal Corp.
Sonex, Inc.
Spartan Books
Spiras Systems, Inc.
Standard Memories—Sub. Applied Magneti
Storage Technology Corporation
Sycor, Inc.
Sykes Datatronics, Inc.
Syner-Data, Inc.
Sys Associates, Inc.
Systematics/Magne-Head—Div. Gen. Instr
Systems Engineering Labs
Tally Corporation
TDK Electronics Company, Ltd.
TEC, Inc.
Technitrend Inc.
Tektronix, Inc.
Teletype Corporation
Telex Computer Products
Tel-Tech Corporation
Tempo Computers Inc.
Texas Instruments
Timeplex, Inc.
Time-Sharing Terminals, Inc.
Tracor Data Systems Inc.
Trio Laboratories, Inc.
Tymshare, Inc.
Typagraph Corporation
Univac—Div. of Sperry Rand
University Computing Company
UTE (United Telecontrol)
Vanguard Data Systems
Varian Data Machines
Varisystems Corporation
Vermont Research Corporation
Versatec, Inc.
Vernitron Corporation
Viatron Computer Systems
Victor Comptometer Corporation
Wang Computer Products, Inc. (WCP)
Wang Labs
Wanlass Electric Co.
Weismantel Associates, Inc.
Western Union
John Wiley & Sons, Inc.
Xerox Corp.—Business Products Group
Xynetics, Inc.

AUTHOR INDEX