ESD-TR-68-143, Vol. IV

EVOLUTIONARY SYSTEM FOR DATA PROCESSING PROGRAMMING SPECIFICATIONS

Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

January 1968

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0254 by Center for Exploratory Studies, International Business Machines Corporation, Rockville, Maryland.)

AD670841

FSC 68-0671/

ESD-TR-68-143, Vol. IV

EVOLUTIONARY SYSTEM FOR DATA PROCESSING PROGRAMMING SPECIFICATIONS

Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

January 1968

COMMAND SYSTEMS DIVISION
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

(Prepared under Contract No. F19628-67-C-0254 by Center for Exploratory Studies, International Business Machines Corporation, Rockville, Maryland.)

FOREWORD

This report presents the results of a study of the specifications for an information system intended to support the design, production and maintenance of large computer programming systems. Called Evolutionary System for Data Processing, or ESDP, it was begun as an internal IBM project in 1965 by the Center for Exploratory Studies of the Federal Systems Division and continued under Air Force sponsorship during 1967 and early 1968.

This work has been performed under contract number F19628-67-C0254 for the Electronic Systems Division, U.S. Air Force Systems Command. The project monitor was Mr. John Goodenough, ESLFE.

The authors wish to express their appreciation for the encouragement and assistance provided by Dr. John Egan, formerly of ESD, and their colleagues Dr. Harlan D. Mills and Mr. Michael Dyer.

This report is in four volumes: Volume 1, System Description; Volume 2, Control and Use of the System; Volume 3, The CAINT Executive Language and Instruction Generator; and Volume 4, Programming Specifications. This report was submitted on January 31, 1968.

This report has been reviewed and is approved.

Sylvia R. Mayer
SYLVIA R. MAYER
Project Officer

William F. Heisler
WILLIAM F. HEISLER, Col, USAF
Chief, Command Systems Division

ABSTRACT

ESDP is a proposed system whose purpose is to acquire, store, retrieve, publish and disseminate all documentation, exclusive of graphics, concerned with a large computer programming activity. Documentation is deemed to consist, not only of final or formally published after-the-fact reports, but of working files, design and change notices, informal drafts, management reports--in fact, the entire recordable rationale underlying a programming system. Maximum attention has been concentrated on the means of acquiring and organizing documentation. Two major, complementary approaches are proposed. The first is called Program Analysis and is a process of _extracting_ documentation directly from completed programs. The second is called Computer Assisted Interrogation and is a process of _eliciting_ information directly from the programmer, through on-line communication terminals. The former provides canonical data about the program's structure. The latter provides explanatory material about all aspects of the program, and in the absence of canonical data, may provide tentative structural information as well. The conclusion of the study group is that ESDP is a feasible concept with present-day technology and that it will materially benefit using organizations in the production of programs and in guiding their evolution as requirements change. Its value will be greater for larger organizations, whose internal communications difficulties tend to cause truly gigantic inefficiencies. Its implementation as a support system for such projects would require a significant quantum of investment in order to produce these benefits and is predicated on the use of a computer system dedicated solely to the use of ESDP.

Volume 4

Programming Specifications

ILLUSTRATIONS

I

SYSTEM DESCRIPTION

1. General Approach to Programming. The general architecture proposed for the ESDP system is that used for Operating System/360-Queued Telecommunication Access Method (OS-QTAM) (see Figure 1). In such a system, terminals communicate with the central processing unit via telephone lines and a multiplexor channel. In the central processing unit, two or more programs are operating asynchronously in separate partitions of high speed memory under control of the OS supervisor.

In one partition, the Message Control Program plus some additional QTAM code dispatches incoming and outgoing messages. The Message Control Program makes use of core buffers (the number and size being specified by the programmer) plus message queue storage on a direct access storage device.

In the other partitions are the Message Processing Programs. These programs perform all the ESDP processing functions. They receive messages from and transmit messages to the Message Control Program via GET and PUT macro commands. When a message has been received, an ESDP controller, one of the Message Processing Programs, must first determine what activity the sender is involved in. For instance, it must recognize whether a message is a response to a question in interrogation or, say, a query. Once this determination has been made, some type of housekeeping, depending on the particular message and activity, is performed to initialize the ESDP functional routines. Program control is then switched to the particular module of programming required to perform the desired activity. These modules interact with the system files and issue messages back to the terminals via PUT commands.

In addition to providing the capabilities outlined above, the operating system for ESDP must be concerned with the following requirements: (1) More than one user terminal may be communicating with any one program module at a time. This requirement may best be met by assuring that the program modules are reentrant. (Note that in our current experimental work we have used PL/I which produces reentrant code.) (2) Different user terminals may be communicating with different program modules at the same time. We feel that this requirement can probably be met by a multi-tasking supervisor such as that now used in OS/360 with Multiprogramming with a Variable Number of Tasks (MVT). This will provide for a primitive form of time sharing by activating tasks whenever an I/O operation occurs, making use of a priority system for the tasks. For the system described, this type of time sharing should suffice, since there should be no periods of long processing, uninterrupted by I/O commands. (3) More than one user terminal may be accessing any one data element at the same time. This will require that some form of data base lockout be placed into the system.

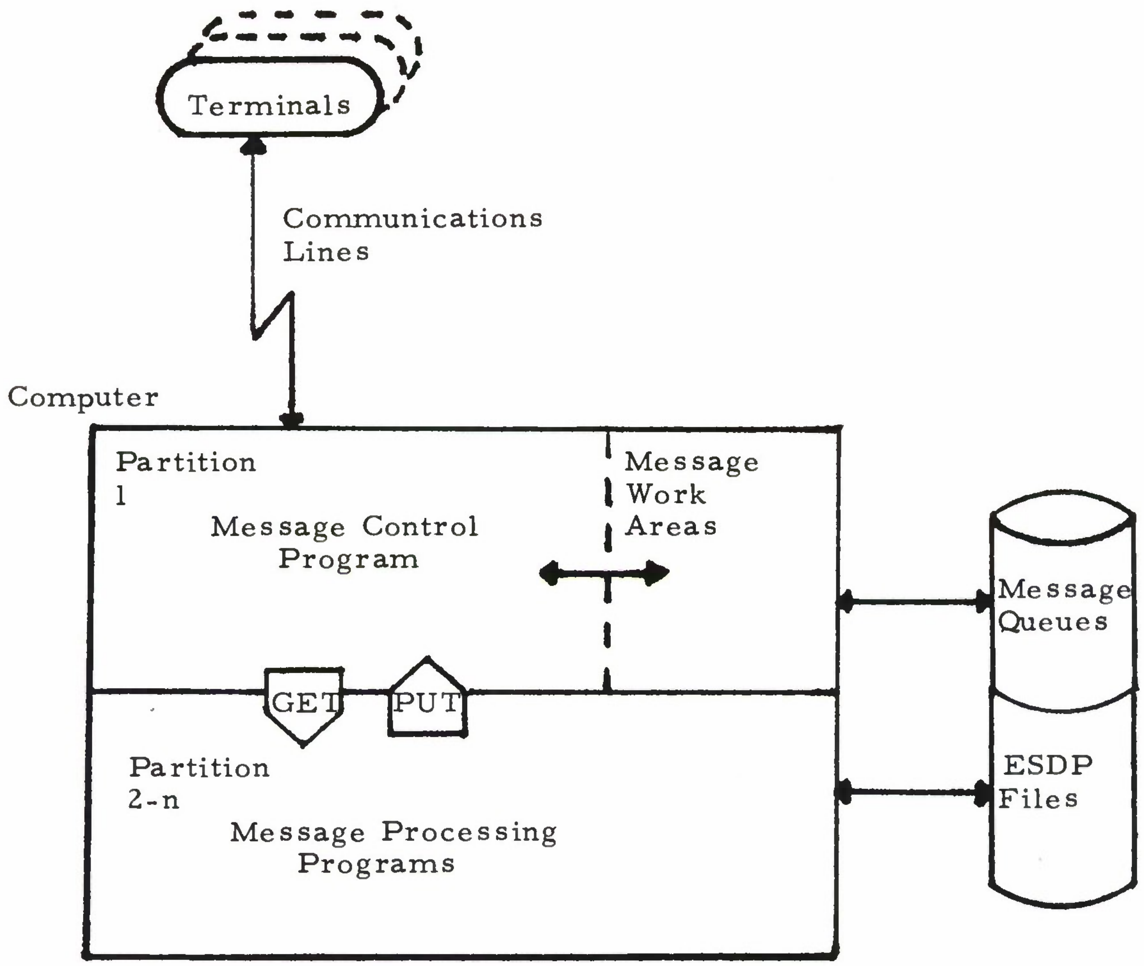

Figure 1. General ESDP System Concept

The general concept of ESDP is for a teleprocessed terminal-oriented system. The terminals themselves should comprise the following:

a. A cathode ray tube display with keyboard entry.

Most of the conversational processes will be performed through this device. Light pen capability and/or vector drawing capability may be desired depending on the need for activities such as production of graphics in the documentation. For the strictly conversational documentation activities, generation of an average-sized character set (e.g., 64 character set including numbers and upper case letters) on the face of the CRT should suffice.

b. Hard copy printer.

It is often desirable to retain a hard copy of that which has been displayed on the CRT. This can be accomplished via a typewriter type printer (without keyboard), the printing being activated by command from the keyboard associated with the CRT.

c. Line printer

Line printing should be centralized so that high volume outputs can be generated in the machine room for subsequent manual transmittal to the requesting user.

d. Terminal polling

A round-robin polling system with priorities such as that used by QTAM seems appropriate. Of course, if inefficiency results, perhaps the priority scheme should be revised so as to be based on the particular activity, for instance, rather than simply the terminal identification.

It is anticipated that during hours when normal ESDP documentation activity is light, other programs can be run that are not under the general QTAM-ESDP set up. Examples of such programs are:

<u>File cleanup programs</u>--It may be necessary to move data on the direct access storage devices in order to reuse space freed via deletion of records. This reorganizing is one type of file processing that might be performed off-line. In addition, there are normal utility functions such as disk copying, disk printing, etc., that could fit in this category.

<u>Pre-processors</u>--There may be some pre-processing desired for the CAINT Executive Language. This is particularly true when debugging macros are to be used. Such pre-processors could operate off-line.

2. Hardware Assumptions. Hardware has not been considered in this study, except indirectly, when feasibility of attaining various objectives was considered. There were, however, some basic hardware assumptions underlying the study. These are:

a. Machine Utilization

A computing system will be dedicated to ESDP.

b. Machine Type

A System/360 computer with Operating System/360 was used for the experimental programming in this project. This choice of hardware, of course, is not mandatory. However, much of the discussion in this report is based on S/360 with OS and, therefore, uses that terminology.

II

DATA BASE

We foresee the need for several files, or, in the terminology adopted herein, file sets. While it would be possible to store most of the information to be described below in one monolithic file, this breakdown recognizes differing frequencies of file modification, different processes to be performed on data, and different means of control of access.

1. Program Description File Set. This file set contains a logical record for each unit of programming. Its organization will bear a close resemblance to the outline of a conventional program description, but there is no permanent standard and it is expected and encouraged that the content and composition of this file will be shaped by the users to fit their own needs.

The major subjects to be covered, in a generalized form of the file are:

- o Identification--of the program, programmer, date, etc.
- o Program Structure--in terms both of the hierarchical structure of the program and of the branching, or control, structure.
- o Data References--the data items named by the program and the nature of their use.
- o Logic Description--both symbolic and natural language descriptions of what the program does, how, why.
- o Management and Status Data--information relative to the program as an item being produced, its schedule, progress, problems, etc.
- o Illustration References--references to flow charts and tables to be composed by ESDP and to be printed with this program description information. Also, references to other graphics, used for illustration, which are not able to be stored within the ESDP computer.

Except for the identification section, for which no amplification is necessary, these items are discussed below in greater detail.

a. Program Structure

This section would contain pointers to related UOP's. There are two general categories of relationship: hierarchical and control. Hierarchical pointers would indicate subordination or superordination, and control pointers would indicate entry points or predecessor and successor UOP's. Entries from, or exits to, label variables would be treated as a special case, with the variable representing a program switch which might be given a form of UOP status. Also contained in this section would be a codification of the type of branch control (whether unconditional, such as a PL/I GO TO; or conditional, such as an IF or DO and the variables that affect the branch. In addition, there would be narrative explanations of the control logic, or pointers to such explanations.

b. Data References

The exact extent to which data documentation should be split or duplicated between the program and the data description files depends on the philosophy of management of the object system. At a minimum, this section of a program description file must list the data elements that occur in the program, and must give the nature of the usage, such as a control variable (a variable that directly affects a branching decision), a computed value(set by an assign or DO statement) or any of a number of other categories of usage. The bulk of the actual description of the data, as differentiated from the codification of the nature of its use, will be carried in the Data Description File Set.

c. Logic Description

The essence of logic description is to tell what the program does, how, and why. But each of these points is subdividable and answerable on more than one level of abstraction. Hence, there may be a large number of individual items contained in this file. It must be emphasized that the logic description is not restricted to narrative data only. Many aspects of a program's function can be codified.

d. Management and Status Data

Perhaps more than any other portion of the documentation files, this will be the one most often changed to suit individual user needs. The information to be collected and the processing to be performed on it is largely a function of the individual manager, who must satisfy his own requirements for information plus those of his higher management and the contractual requirements for system documentation.

The obvious kind of information to be collected has to do with progress in meeting schedules and milestones, manpower assigned, problems being met or anticipated, and budget data. Mostly, the information will be collected by interrogation, but some, such as number and dates of compilation, program lengths, etc., can be acquired automatically.

e. Illustration References

We may group illustration material into three classes: flow charts or tables that are required according to the system documentation plan, flow charts or tables volunteered by the programmer or file documentor to augment some aspect of his narrative, and other illustrative material that is volunteered, but is not in flow chart or table form. The restriction on flow chart and table form arises, of course, from the planned use of standard graphic programs that will produce these configurations easily, but cannot handle the full range of graphic input. Thus, if a programmer wishes to input a logarithmic graph, or a diagram of two aircraft on a collision course, he will probably have to draw these in the conventional manner, but include in the machine-stored documentation, a reference to the illustration copy.

In the program description file set will be stored only pointers to the detailed illustration information, whether or not stored within ESDP. We recommend this separation because these files will be large, and, while they may be updated whenever the program is, they will rarely be subject to information retrieval searches in the same way as detailed data in the remainder of the file. A user may want to retrieve the information that is displayed on a chart, but he will not normally want to retrieve the detailed FLOWCHART instructions that organize the display.

2. Data Description File Set. The recommended approach for documentation of data files is to create a data description record for each file or structure used in any program, with as many working records as needed to give each user a chance to document the data the way he prefers. Another version of the descriptive record will be created and maintained by an authorized COMPOOL, or data base, controller in whom will be vested authority to make final decisions on data definitions and attributes, and who can delete working records at will. His expected mode of operation, then, should be to review his data items periodically, look at the conflicting descriptions or requests of the individual programmers, make his decision on which version to accept or declare, and put the final decisions into the permanent record.

Provision can be made, using the dissemination services, to promulgate the data base controller's decisions immediately to all programmers concerned.

The organization of the data description record will be similar to that for a program. It will have the following major headings:

- Identification
- Element description--the narrative and other information about the item and its use.

- Structure--primarily, this is used for higher level structures in order to define the subordinate structure.

- Using program references--pointers, possibly some supporting text on what effect the particular using program has on the data.

- Illustration references

3. Program File Set. This file set will contain the text of the programs being documented. In addition, consideration will be given to storing, within this file set, program change information, separate from the program, itself. This will permit a record to be kept of all changes, and this will permit programmers to make changes either to the latest version of a program, any previous version, or both simultaneously. The complete text of any version of the program could be retrieved on request. Another possible class of information for this file set is partially reduced program analysis data. This would be intermediate output, produced during a program analysis run, which could be saved to reduce the time required to process a change to the program.

4. Graphic Coding File Set. We recommend the use of a program for the automatic production of flow charts and tables. In some systems, such as FLOWCHART [1], graphics are assembled by the issuance of commands on how to build them, in a manner similar to computer programming. The Graphic Coding File Set would contain these instructions.

The graphic files may be updated separately from the program or data files they illustrate, but this form of updating should probably be restricted to changes in layout. Changes in content or structure should be keyed to changes in the data or programs being described, although the initiative for a change may originate with either a program or data description change or with a graphic change.

5. Publication File Set. This file set will contain partially processed documentation, taken from any of the other files. The preparation of copy for publication can be a time-consuming process. Hence, partially edited material should be retained in machine readable form for reprinting or for selection for inclusion in differently-organized documents. This form of storage is used by the IBM Administrative Terminal System [2], a text processing system, a successor of which is recommended for use in ESDP.

6. <u>Instruction Course File Set</u>. Instructional material produced through ESDP plays a dual role. It is used in its own right as a system program, but it is also a form of documentation and changes in it must be keyed to changes in the programs or data being taught. Hence, instruction courses can be stored as programs, but must have appropriate pointers back and forth to the documentation from which they were derived.

7. <u>Dissemination File Set</u>. These files will contain the profile and distribution lists needed to operate the internal ESDP dissemination system on documentation and changes thereto.

8. <u>Index File Set</u>. These files are those indexes and inverted indexes used by the information retrieval system to carry out its functions. These are also dynamic files, which are subject to frequent change as the documentation files change.

9. <u>Buffer File Set</u>. Buffer files are dynamically created files and are for use by the information retrieval system.

III

PROGRAM ANALYSIS

1. <u>General</u>. A Program Analysis (PA) program has been produced by IBM [3] as part of its own internally sponsored ESDP activities. This program accepts input in PL/I, OS/360 Job Control or OS/360 Linkage Editor languages and compiles a canonical or structural data file descriptive of the hierarchical and control structure of the programs and their usages of data.

Source Code Analysis is performed by a set of compiler-like analyzers which are oriented to a particular language. The number of analyzers is dependent on the make-up of the user's programming system. The role of each analyzer is identical regardless of the language, namely to map source code into the UOP coordinate structure and generate the data records associated with it. In this way, each analyzer, which is necessarily language dependent, can effect a common interface with the system.

Three analyzers have been written, for OS/360 JOB Control Language (JCL), OS/360 Linkage Editor Language, and OS/360 PL/I Language. This sample was selected to permit experimentation with programming systems written primarily in PL/I, of which the analyzer, itself, is an example. We note again that the treatment of run-time languages (e.g., JCL) as programming languages is a critical point since it is key to the automatic analysis of system-wide interactions.

Current compilers and assemblers now generate source code listings and cross reference lists for data variables for a single program at compilation time. However, this is ordinarily the extent of their automatic capabilities. Additional programming information on program interactions within a larger system, rationale behind program logic and program groupings, data flow through the system, and so on, are necessarily based on interrogation-acquired documentation.

The analyzer parses the source program into elements called Units of Programming (UOP). The current program produces UOP's at the following levels:

JOB

LOAD MODULE

SOURCE MODULE (Compilation Unit)

CALL MODULE - Procedure Block

GROUP - BEGIN block
DO group
IF compound statement
ON compound statement

SEGMENT

For each UOP in a structure, a data record is created which contains the appropriate structure, logic and data usage information. This structure and logic data of the individual UOP records is also the mechanism for creating the total structure of a program system or any of its major components.

To make the programmer aware of how his program is structured, a revised program listing is generated, which graphically depicts the coordinate structure and labeling for this program. This revised listing is useful, not only as a guide to the files, but also as a picture of program execution which may easily become obscure in the compiler generated listing, particularly with free format languages where multiple statements can be strung together in a single print line.

System-wide interactions of a program can be obtained through the automatic analysis of the Object Module generated by the compiler and the JCL deck that would be written for execution.

Object module analysis yields information regarding program interaction (external procedure in PL/I terms) in a system. This process consists, basically, of extracting symbols from the external symbol dictionaries of all the object modules involved in a given linkage editor run, detecting module cross-references and, discriminating between data references and branch references.

This information is critical when sets of programs are linked together and manipulate the same data, since this is the source of most problems and delays during integration of programming systems where the various pieces were written and debugged by different people.

Within OS/360, the execution of a program would require a JOB Control deck or program. The analysis would equate, within the UOP records, the file declarations at the JOB level with all references to these files down to the SEGMENT level. In a more complex case, where condition codes and multiple job steps were defined, this same correlation of program units and data usage would have added significance.

2. Operation of Program Analysis. The analysis of PL/I code is performed in several phases. In Phase 1, PL/I source code is read in, and blanks, comments, and constants are eliminated. The remaining characters are translated through use of a translation table. The general effect of this translation is to replace the source language string with numeric codes in such a way that

alphanumeric strings are grouped in the higher number codes, special characters in the lower number codes, and operation codes in the middle. This is done so that in future processing, simple limit testing can be used on the codes to determine the type.

An output string is organized in which each statement is given a statement number.

The statements are scanned and whenever labels, file names, parameters, or condition names are encountered, a dictionary entry is created, and a pointer to the dictionary entry is stored with the statement.

When UOP defining statements (e.g., DO, BEGIN) are encountered in the scan, entries are made in a parsing table. Then, when statements are encountered that end UOP's (e.g., END), the table is searched to determine which entry is closed. The table then contains the statement numbers defining the limits of the UOP's.

At the completion of Phase 1, the parsing table has been filled, and the dictionary has been partially filled.

Phase 2 reformats the source text, indicating the parsed units and statement numbers. The units are indicated in such a way as to ease reading.

In Phase 3, DECLARE statements are analyzed. This is done using an array of attribute masks. Each data attribute is represented by a 32-bit mask (row). Each element, A(i,j) represents the interaction of attributes i and j. If A(i,j) is one, then the two attributes can co-occur. Zero means that they cannot co-occur. For instance, EXTERNAL can co-occur with FIXED but not with INTERNAL. Each attribute is looked up in the table of masks and all of the masks are AND'ed together. The result is a 32-bit string with ones representing the attributes of the DECLARE'd data. Note that by starting with the assumption that all attributes apply and then ruling out impossibilities, defaulted attributes are also depicted. Scope tables are generated for the data and these plus the attribute masks are added to the dictionary, which is now completed.

At this point in the Program Analysis process, two internal tables have been built--the dictionary and the parsing table. Their formats are described below.

a. Dictionary

(1) Bytes 1-2 are reserved for a hash chain during the first three phases of analysis and for a data type code during the latter phases.

(2) Bytes 3-6 contain the scope data for the entry in terms of procedure and statement numbers.

(3) Bytes 7-8 contain for structures, pointers to structure elements, and for labels, the statement number of the label declaration.

(4) Bytes 9-28 contain the identifier as it appears in the source code.

(5) Bytes 29-32 contain a bit table that defines the unique attributes or characteristics of the entry.

(6) Bytes 33-40 are a set of offset values that point to the overflow area. Note that certain PL/I attributes carry value information, e.g., precision, bounds of dimensions, file environments, etc. This value data is stored in the overflow area and the offsets are used to delimit the start and stop of various values.

(7) Bytes 40-119 contain any values associated with attributes.

Figure 2 illustrates a typical dictionary entry.

b. Parsing Table

To delimit UOP structures and keep track of UOP nesting, this table is generated during the analysis. It also is delcared as an array of bit strings where each element represents a single UOP. The format of each element is as follows:

(1) Bit 1 - status switch used to determine if UOP has been closed.

(2) Bits 2-9 - UOP level code where code runs from 1 - 6, corresponding to JOB level to Segment level.

(3) Bits 10-73 contain the procedure name or label on the including procedure.

(4) Bits 74-85 contain the statement number of the first statement in the UOP.

(5) Bits 86-97 contain the statement number of the last statement in the UOP.

(6) Bits 98-106 contain a dictionary pointer to the label associated with the UOP, if any.

Figure 3 illustrates a typical parsing table entry.

	Hash Chain	Scope	Structure Pointers	Data Name	Attri- butes	Table of Offsets	Over- flow Area
Byte	1	3	7	9	29	33	40

Figure 2. A Typical Dictionary Entry.

	Switch	Level Number	Including Procedure Name	First Statement Number	Last Statement Number	Dictionary Pointer
Byte	1	2	10	74	86	98

Figure 3. A Typical Parsing Table Entry.

The string generated in Phase 1 is read in Phase 4, and a new string is produced that is completely coded. All identifiers are replaced with dictionary pointers.

Phase 5 determines data type for all data in the dictionary and adds a data type code to the dictionary.

Phase 6 reads in the parsing table and reads in the program statements, one at a time. From these it generates the UOP records and with the additional input of the dictionary it generates the trailer records. These are written out on tape.

The JCL cards are used by the JCLSCAN Program, and each card is examined to determine if it is a JOB card, an EXEC card, a DD card, or other. Cards in the other category are immediately rejected. JOB cards are further examined for condition codes at the JOB level. If they exist, they are stored on an analysis list.

For EXEC cards, the program stores the job step name in the analysis list and then determines if the name refers to a catalogued procedure. If it does, the name is marked as job level. If it does not, the name is marked as load module level. The EXEC card is then checked for JOB step parameters, and if there are any, they are stored in the analysis list. The same process is followed for JOB STEP condition codes.

For DD cards, the DSNAME is stored in the analysis list along with any disposition parameters.

After all of the JCL cards have been read, the analysis list is further processed, the process varying with the type of JCL statement.

JOB - The UOP name is extracted from the job statement label field. The entry and exit portions of the UOP are marked, and if condition codes exist, subordinate UOP are marked as exit points.

DD - A data reference entry is made in the UOP for the DD.

EXEC - JOB STEPS become subordinate units to the JOB UOP. The UOP names are the JOB STEP names. If JOB STEP parameters exist, a data reference entry is made using a dummy name. If JOB STEP condition codes exist, the subordinate transfer table is marked accordingly.

The Linkage Editor Analysis Program (LEAR) begins by reading from the primary input stream. A test is made to determine if the first entry in the stream is a linkage editor command. If it is not, the entry is processed as an object module. If it is, another test is made to determine if the command is an INCLUDE statement. INCLUDE statements effect readings from secondary input streams. All other command types are ignored. Object module processing continues in the primary stream until an INCLUDE is found. Then, processing shifts to the secondary stream. In the secondary stream, the first column of the card image is checked for a blank (indicating command), a 12-9-2 punch (indicating an object module), or any other (indicating a load module). The first two are processed as previously discussed and the third (load module) causes a load module subordinate unit entry to be established.

Once all of the linkage editor object modules, load modules and commands have been processed, a UOP record is formed. This UOP is in the same format as a PL/I UOP.

3. Additional Requirements. There must be added to the program analysis implementation an incremental analysis capability. When a programmer makes a change to an existing program, he should not have to run the entire program through analysis again. This process now takes an amount of time on the same order as a full compilation, hence in a large system it could become a significant drain on computer capacity if repeated often. Instead, the approach recommended is to have ESDP store the latest copy, and let the programmer make changes by use of ADD, CHANGE, and DELETE commands, treating his stored program as a file. In this way, PA need only analyze the changes and make minimal modification to the canonical data file, and new interrogations can be initiated only on those portions of the program affected.

More detailed information than that produced by the present PA program is needed for classifying the manner in which data labels are used or referred to within the program text. For a variety of reasons (e.g., making up more detailed interrogations, assisting in test planning, assisting in debugging, and providing better cross indexing of documentation), we feel that data usage should be classified in as much detail as possible. Furthermore, the information desired is available through the program analysis function, but currently is discarded rather than saved (this is also true of compilation). A hierarchical classification code should be used for each appearance of a data label. This code should reflect whether the item is changed by this usage or not; whether it is changed by being assigned a new value or having a new value read in: whether it is used without being changed; whether when used in an assignment statement, it is used as an item in itself, or an index to another item, a control item, etc. A first approximation to such a classification system is given in Figure 4, which also appeared in Volume 1.

1. Context of Appearance

- 1.1 Assignment Statement
  - 1.1.1 Computed Value
  - 1.1.2 Argument

- 1.2 Control Statement
  - 1.2.1 Variable I/O Command
  - 1.2.2 Branching or Transfer Command
    - 1.2.2.1 Argument or condition statement (IF, ON ...)
    - 1.2.2.2 Iterative Control Variable (DO)
      - 1.2.2.2.1 Initial index value
      - 1.2.2.2.2 Increment
      - 1.2.2.2.3 Maximum value or limit
    - 1.2.2.3 Variable address

- 1.3 Subroutine/Function/Macro Calling Sequence
  - 1.3.1 Transmitted to SR/Function/Macro
  - 1.3.2 Received from SR/Function/Macro

- 1.4 Data Declaration Statement (or other non-executable statement)
- 1.5 Input/Output
  - 1.5.1 Input
    - 1.5.1.1 Input Control Variable
    - 1.5.1.2 Data Element read in
  - 1.5.2 Output
    - 1.5.2.1 Output Control Variable
    - 1.5.2.2 Data Element written out or transmitted

2. Change Status

- 2.1 Value Changed by Containing Statement
  - 2.1.1 Value Directly Assigned by Assignment Statement
  - 2.1.2 Value Directly Changed by DO Statement
  - 2.1.3 Value Directly Changed by Variable I/O Statement

- 2.2 Value not Changed by Containing Statement

3. Structural Role

- 3.1 Data Element is a Structure or Array

- 3.2 Index or Subscript
  - 3.2.1 VALUE OF AN Index
  - 3.2.2 Element of an Index Term

- 3.3 Scalar Item

Figure 4. Classification of Data Usage by a Program.

Another aspect of program analysis (or possibly of information retrieval) to be borne in mind is that, as the documentation files grow large, there will be inevitable errors, such as programmers misnaming programs, submitting the wrong version of a program for analysis, entering changes incorrectly (resulting in an actual program that differs from what the author thinks it is), etc. These are normal mistakes of any programming project and, in a purely manual system they can be tolerated and relatively easily reversed. The documentation file system and the program analysis system must be so designed as to anticipate such errors and, while it is not ESDP's responsibility to detect them, it should be possible within ESDP to correct them with minimum difficulty.

IV

CONVERSATIONAL PROCESSING

A thorough description of the conversational language and processing required is contained in Volume 3 of this report. Programming of the conversational routines represents a considerable share of the entire ESDP implementation effort. The scope, techniques and other aspects of the programming required should become evident from the discussion of the conversational activities and language contained in that volume and therefore will not be further discussed here.

V

INFORMATION RETRIEVAL

1. ESDP Files. The files handled in the ESDP system include the following:

a. Program Description File Set

This file set contains a record for each UOP in the object system. The information in the file may be derived through program analysis, interrogation, or both with the source being identified.

b. Data Description File Set

This file contains a record for each Unit of Data (UOD). Here, the information is obtained through interrogation only. UOP and UOD are linked via pointers since the data are referenced in UOP.

c. Index File Set

A file is build of keywords indicating for each the UOP or UOD and IEN associated with the key word. The key words are extracted automatically at the time that a response to a question is entered during interrogation. At that time, the UOP name of the UOP currently the subject of the interrogation and the IEN associated with the question are appended to all key words in the response.

d. Buffer File Set

Provision is made in the ESDP concept for general file handling capabilities. Programs that interpret file format tables for file accessing will be included. In addition, file building may be done on line as well as off line. The intended use of the special files is as personalized subsets of the ESDP data base. It is anticipated that this feature would be heavily used by system managers to create, update, and search personalized management information systems.

2. File Building and Maintenance. Creation and modification of UOP records and UOD records are planned in ESDP to match the changes in the object system of programs. Records are created whenever the system becomes aware of a new UOP or UOD. This information may be acquired in any one of a number of ways:

a. Source Code Parsing

Program Analysis creates UOP's by parsing source language code. UOP's created are named either by program label or by a combination of containing UOP name and statement numbers.

b. Source Code References

References may be made in the source code to UOD or other UOP not subject to Program Analysis. The appearance of these references in source coding will cause the creation of the appropriate records and names will be taken from the source code.

c. Interrogation

UOP or UOD may also be named by the programmer at a console in the interrogation process. This can occur during design interrogation or during interrogations performed after the object system program has been subjected to program analysis. Naming of these UOP's and UOD's is a simple process since the programmer assigns these.

Whenever changes to source code are submitted for analysis or incremental interrogations are processed, changes to data items in existing UOP or UOD records are likely to take place. The changes can take the form of ADD, DELETE, or REPLACE (DELETE and ADD). The way in which the system handles the file updating will depend on the data elements to be changed and the manner in which the requested changes are entered into the system.

ADD'ed data items derived from interrogation may be handled directly since a full record is created for each UOP or UOD whether or not all of the data item fields contain information. Therefore, an ADD amounts to storing the new information into an appropriate position in the core-resident image of the UOP/UOD record, and rewriting the record to the disk file. Cross references are added in the normal manner. DELETE's and REPLACE's present a more difficult updating problem however. Again, data may be deleted in the same manner as it was added above. In this case, however, cross references must also be updated. For instance, assume that a programmer wishes to delete textual information associated with a given IEN. The text may be deleted, but keyword references to the text must also be. This will be done by performing a second keyword extraction on the text to be deleted. The keywords extracted will then be used as search arguments for the keyword index records so that the appropriate IEN pointers may be deleted.

The general concept for ESDP file updating as a result of changes to source programs is to rerun Program Analysis on the UOP containing the changed UOP. Old UOP records will not be erased at this time. Through the reconciliation process, information associated with the old UOP records will be linked to the new UOP records. When the reconciliation has been completed, the old records will be deleted. Keyword references must be updated during the reconciliation process. If text from an IEN of the old version of a record is to be moved to another IEN of the new version, the keyword updating amounts to changing the IEN pointers in all of the appropriate keyword records. If new text is typed in, keywords are extracted in the normal fashion. For all deleted records, the keyword deletion is performed as in the case of deleted text through incremental interrogation.

3. Keyword File. An experimental keyword extraction program is now being tested. This program operates as follows:

(1) Responses to questions are subjected to keyword extraction under the control of the CEL Program.

(2) Responses are edited to eliminate deleted lines, to eliminate deleted characters (backspace and retype), to eliminate carriage returns, and to convert all letters to upper case. This is done to eliminate mismatches in the keyword list. For instance, "Computer" without such editing would not match with "computer" and similarly carriage return characters, backspace characters, or delete characters will eliminate any possibility of an exact match.

(3) Each word in the response is compared with words in a common word list. Common words are not stored as keywords.

(4) Each keyword (i.e., not common word) is stored and is tagged with the IEN associated with the question to which this is a response. If the keyword is already recorded, the IEN is added to a list of IEN's in which the word appeared. In addition to the IEN, the keyword could be tagged with its position within the response. This would enable subsequent retrieval based on position of keywords in a response.

4. Searching. Information in ESDP is indexed in four ways:

a. Program Element

One index to a piece of data is the particular element (UOP, UOD, etc.) with which it is associated. This information is obtained through interrogation for design documentation and through program analysis for final documentation.

b. Keywords

Another index is the keyword index. The keywords are extracted automatically from responses to interrogation questions.

c. Data Names and Labels

These are character strings used in the program or the program design being documented. They, too, serve as indexes to the UOP or UOD records.

d. Hierarchic Codé

ESDP employs a hierarchical coding system and attaches a code number to each element of data. This number is called an

Information Element Number (IEN). The structure of these numerical codes is intended to classify any data collected by ESDP about an object system of programs.

Searching of the ESDP files is requested from terminals or ESDP programs. The query language is basically the same subset of PL/I as is used for executive programming. Again, the IF statement is the core of the subset language. The user may express complex Boolean relationships as the conditional statements. For instance.

```
IF (A > B) & (C = '1') (A + C ¬= '7')
```

is an example of a conditional statement. Here, A, B, and C are data base items, uniquely identified. The system must be capable of so identifying data base elements and will do so through use of symbol tables.

An additional feature of ESDP information retrieval is the ability to specify a file as the disposition for retrieved data. Thus, the user can dynamically create files from information in other files. He may perform cyclic searches by retrieving data, placing it in a buffer file, and then using the retrieved elements as search criteria for another search.

Cyclic retrieval is defined as the use of information retrieved from one query as part of the statement of a subsequent query to the same or a different file, so that a cycle of query, retrieval, query based on retrieval data, retrieval, etc., can be set up.

Dynamic file creation also allows for creation of small "customized" files for use only by a particular individual or organization. These files would be organized for the particular application at hand.

Outputs of results from searches are highly flexible in ESDP. A user may specify (1) mode of output (printer, console, etc.), (2) data manipulations (sorts, totals, other calculations, etc.), and (3) formatting (page widths, spacing, titles, etc.)

The information retrieval function should be designed to operate both as a subsystem of other programs and to service directly queries submitted through on-line terminals or through a background, or deferred, job queue.

Every effort should be made to integrate the query language with the Interrogation/Instruction executive language, using a common syntax and a common interpreter or compiler. Probably, a compact form of query statement will be used by the programs, with the human-written form translated into this compact form. Programs' calling information retrieval would directly fill in this table, somewhat as a calling sequence. The human user then has the options of writing out his query directly in this compact language if he wishes and is adept enough, writing out the query in a more natural form (not natural language, but a programming-like language) or building his query gradually through a computer assisted, conversational process.

In regard to performance, while specifications have not yet been developed, it seems that the following are required:

- Records must be retrievable on the obvious characteristics which are usually unique identifiers: address, sequence number within a file, value of a key or sort field.

- Records must also be retrievable on the basis of Boolean combinations of these or other record attributes, each attribute (probably) being able to be stated as one or more relationship statements, as SALARY = 10000 or AGE $<$40.

- Individual items, fields, arrays, sub-records, etc., can be specified as the information to be retrieved from a record--the entire record need not be retrieved in response to a query. Thus, the burden of extracting the exact information needed from a record is placed upon the retrieval system, not the calling program.

- Information called for may be ordered to be held in a buffer or temporary storage area for later reference. In particular, this requirement is imposed to make cyclic retrieval possible.

- The requestor, whether a person or a program, may specify the recipient of the information, which need not be the requestor. In other words, an IR system user may call for the retrieval of information and its presentation to some other person, output device, or program. Thus, the retrieval system can be used as an internal message handling system.

In ESDP the user will specify explicitly what are usually implicit in a query--the precise data to be retrieved (not the entire record containing what is wanted), the place to store it (if other than the printed page or a CRT console) and the addressee, who is usually the requestor.

It should be noted that the requirements for a system responsive to both human requestors and programs, with one of the using programs being a query acquisition program that communicates with the human user.

A feature that will be required of ESDP will be to index narrative interrogation responses to permit access by the retrieval system on the basis of response content. There are several reasonably well established techniques for doing this. One is to use a list of "common" words (articles, the forms of the verb to be, etc.), delete these from responses, truncate the remaining words at five or six letters and use them as a keyword index. Alternatively, a dictionary of system terms can be built and this used to identify words in a response that ought to be in the index to the response. This list must be constantly modified to be sure it is up to date. Another automatic technique that might be useful is to require that a special character precede or follow a data element, program name, or other system label when used in text. In this way, any cross reference in a response can be readily identified.

More generally, the logic of computer assisted interrogation gives us the following information about a narrative documentation item, before it has been elicited from the programmer:

- o Subject of the question--name of UOP or data element, particularly aspect being questioned.
- o Structural relationship, for cross-referencing purposes, with other programs or files.

These items, combined with keywords extracted from the response, give the potential of a very rich keyword index for use in querying or in automatic dissemination. The same items can be used to form an index in each published report. These indexes would, of course, be automatically modified if the basic documentation were modified, either through interrogation or program analysis.

We anticipate that some number of standard queries will be previously written and invoked by the user as he needs them. Some of these queries may be complete as stored and some may need completion or assignment of values to parameters through interrogation.

This type of standard query should be quite easy to implement. The majority of queries, however, will be unanticipated. These will be processed through an interpreter program designed especially to operate on queries expressed in the CEL. The interpretive approach is dictated since compilation of queries cannot be performed rapidly enough to permit an efficient on-line system.

Many information retrieval queries will be of a form in which a single data file is used and a single IF statement is sufficient to decide upon record selection. Often, the key of the record will be given so the desired record may be immediately retrieved. If the key is not given, the implication is that each record must be examined for its compliance with the query, a process considerably shortened by the use of inverted indexes, if they exist. Checking for the existence of these indexes, and making use of them, is a function of the interpreter.

In a typical query, the program will have been written in skeleton form, and the remaining data is acquired at the time of invocation. The items acquired are:

- o Record selection criteria--a single IF statement, although containing any number of clauses.
- o The "THEN" functions--what to do with a selected record, e.g., RETRIEVE items A, B, C, retrieve A to B(1,I) ("retrieve item A and place it in record I of Buffer File 1.")
- o The "ELSE" functions--iteration logic will be built into the original, but the user can add functions. He may, for example, choose to retrieve on the basis of a false IF condition.
- o Processing of retrieved output that has been stored in a buffer, eg., SORT (B1) on B(1,1), i.e., sort file B1 on the first field of a record.
- o Additional commands, such as DEFER, SAVE, etc.

VI

PUBLICATIONS

There are several classes of publications, with fairly important economic differences among them. There will be a class of output for design notes and change notices. This class will be characterized by large volume and high frequency of issue, especially early in a system development cycle. These documents must be disseminated fully and rapidly. There is no great need for many of the niceties of publication that are useful in other forms of documentation. They can be printed at the consoles used by the recipients, or they can be batch printed on a high-speed, centralized printer and disseminated through the organization's regular internal mail system.

As design and production progress, programmers, designers and managers will want fairly complete documents on their own and closely related programs and files. These will be used for ready reference, and possibly for making notes to be used later, during interrogations. This class of documentation is characterized by larger documents of lower frequency of issue, but probably benefitting from more careful physical layout and printing. They will, of course, change often, but many times the holder of such reports can attach a change notice directly to this report copy, or make a hand-written note thereupon. He need not reproduce the entire report every time there is a change to it.

A third class of documentation is the formal documentation normally produced at the end of a project, or for major progress or milestone reports. These are printed much less often than the others, but require many printing features not always available on computer-generated documents.

It appears, at this point, that the logical capabilities represented by existing programs, such as FLOWCHART [1] and Administrative Terminal System [2], will handle most of these documentation problems.

ATS offers all needed features except ability to handle graphics. It offers a much wider choice of type fonts when printing at a terminal with changeable type elements, and the ability to underline text. Variation in type fonts for programming documentation is useful to help distinguish, for example, between labels or data names and normal English usage, as SPEED is a data item, but <u>speed</u> is a rate of motion.

AUTOCHART [4] enables the entry of flow charts and tables. It is designed to accept manually prepared input, hence should be able to interface smoothly with the interrogation processor. The designer does his own flow chart layout. The compensation for the extra work of doing this is a compact chart organized in the most meaningful way, to the author. Tables and charts can be modified without complete regeneration, using an updating interrogation, as in CAINT.

VII

FILE PROCESSING

1. Requirements. From an operational viewpoint, ESDP imposes the following storage/retrieval requirements:

a. Data stored on or transferred to bulk storage must be directly accessible to satisfy a broad range of user query requirements and data storage requirements from on-line consoles.

b. Large data base processing capabilities must be provided in order not to restrict the size of user programming systems.

c. Evolutionary file growth must be accommodated since at the outset of the programming development cycle the ESDP files are empty and evolve as the user's programming system develops.

d. Highly variable record lengths must be allowed since these are dictated by the varying characteristics of the programs comprising the user's programming system.

e. The processing cannot rely on predetermined knowledge of the distribution of search keys, used in accessing data, since these are dictated by the symbol coding conventions adopted for the user's programming system and by his natural language responses to interrogation.

f. Certain files are directly related to others. For example, keywords are related to the UOP in which they were used. Therefore, access to one may necessitate access to the other.

The ESDP file processor addresses these requirements and attempts to provide a solution that effectively handles each requirement within the total context. While this may not be the optimum solution for any given requirement, when considered by itself, it does cope with the totality of requirements in an effective fashion.

A total file management or information processing system was not considered to be an appropriate development based on ESDP requirements. The preferred approach was to develop a set of generalized modules to perform discrete functions which would be usable throughout the ESDP system.

Experimental versions of the file processing routines have been written in the PL/I language. The physical file processing uses the Basic Direct Access Method (BDAM) [5] through PL/I. All data sets are physically organized by regions, where a region is defined as a unit of storage, equivalent to a disk track. This equivalence is based on the current PL/I implementation and may vary as other storage devices are supported in subsequent implementations.

2. Rationale for ESDP Approach. The following discussion is limited to accessing techniques for files where data or records must be directly accessed. It excludes techniques which rely on sequential data organization and on a total file scan. While the latter have application in certain classes of retrieval problems, this is not the case in the ESDP system, since we are dealing with a large data base and a non-batched query/retrieval environment.

The accessing problem is one of uniquely locating each unit of data within a file. Two general techniques can be used to perform this location function; namely, table look-up and randomization techniques.

Let us consider randomization techniques first. These are based on some arithmetic manipulation of the character codes of the name or key for the data to be located. The manipulation results in an address for the peripheral storage device at which the desired data is stored. Numerous techniques are available for randomizing or manipulating the character codes. Each is effective in a given application because it is tailored to the peculiar characteristics of the names or keys used in the application. However, no techniques exist which guarantee a unique transformation in every case. To handle the problem of non-unique transformations, so-called 'chain' processing techniques must be employed (e.g., hash tables). The effectiveness of a technique in any given application is dependent on how well it restricts the size of chains and on the overflow procedures adopted for chain processing.

For the ESDP system, randomization techniques were rejected as the bases of the file accessing mode. First, as noted earlier, the names or keys used in ESDP file accessing are dictated by the symbol coding conventions adopted for a user's programming system and his natural language responses to interrogation. They cannot be predetermined. No known randomization technique exists which can produce satisfactory results, given any key set.

Second, randomizing techniques are useful only for a single access path to file data (i.e., access through a single key set). Because of the nature of the data in the ESDP files, multiple access paths must be available to the same data. Thus, table look-up techniques would be required to handle the secondary key sets and access paths.

Randomizing techniques are more effective in loosely packed file situations. Efficiency drops sharply as denser packing is used. The resultant increase in storage requirements cannot be offset by comparable table look-up storage requirements. Thus, this technique would unduly tax storage requirements. File maintenance also becomes a problem if extensive chains develop.

The alternative to randomization is some form of table look-up which is the method employed in the ESDP file processor. Table look-up techniques employing indices have been used in many other systems and are the basis of the index sequential access method in System/360. Essentially, a table entry is created for each name or key used in accessing a file, and an address of the appropriate file location is stored with the key. When the table is searched, the required storage key can be obtained directly. Various searching algorithms can be used depending on the ordering of the keys in the table.

The most efficient searching techniques require an ordered (typically alphabetic sort order) table based on key characteristics. A major problem arises with these techniques when applied to evolving tables or indices. Either strict order is maintained by physically rearranging the index when new entries are inserted or chaining techniques are used. With the latter technique, new entries are not inserted in sequence but stored separately and a reference inserted at the required point in the sequence. To avoid extensive chain processing, file maintenance of the indices is periodically required, with the frequency of the period dictated by the index growth pattern.

To avoid this maintenance and reorganization problem, the ESDP file processor uses a different technique for index building and searching, which is a take-off on existing list processing ideas.

In the ESDP file system, the index is treated as a group of entries which are physically strung together into a list, not necessarily contiguously, and which are logically ordered or sequenced by the use of pointers or address indicators which are appended to each entry. Because of this uncoupling of the physical and logical ordering of the index (or any list), we can eliminate the index reorganization problem, and with some other simple techniques, the index maintenance problem.

A binary tree structure was selected to permit efficient search strategies, based on binary search techniques. The form of the index entry (or structure node) adopted for the ESDP case is shown in Figure 5. Here:

a. The Index Key Field contains the key or name used to access file data. This field contains such elements as the names of the (1) Units of Programming (UOP); (2) data variable names; or (3) descriptor terms (i.e., keywords).

b. The Low Sequence Pointer contains the address of another index entry whose key is lower in sort sequence than the key of the record being examined. Similarly, the High Sequence Pointer contains the address of a record whose key is higher in sort sequence than the key of the record being examined.

c. The Data Field contains any additional data that is desired to be stored in the index. For ESDP, this field could be

Index Key Field	
Low Sequence Pointer	High Sequence Pointer
Data Field	

Figure 5. Index Entry

used for citation lists, disk addresses and allocation and addressing controls.

This general form was defined to permit the implementation of a single program to perform index building and searching of a variety of indices, each of which had a different specific organization. As typical in list processing, an initial pointer or 'anchor' is maintained that points to the first index entry or head of the list.

3. Prototype ESDP Index Implementation. Typically, list processing techniques have been applied to lists which can be maintained in core memory. For the ESDP problem, the file sizes and index requirements are too large to justify core resident indices; thus, some different techniques had to be employed which could operate with a disk resident index. First, indices were segmented and these segments were the units for storing and retrieving from disk. The selected segment size was set at the track size of the disk unit used. The following PL/I structure declaration defines the segment format used:

SEGMENT FORMAT

```
1  STRUCT,
2  ID BIT (8)                 /*Index Identifier */
2  ANC bit (8)                 /*Next available entry pointer */
2  MINSTRUCT (N),
     3  KEE CHAR (M),         /*Index Field */
     3  LOW BIT (16),         /*Low Sequence Pointer */
     3  HIGH BIT (16),        /*High Sequence Pointer */
     3  ENT CHAR (O),         /*Data Field */
     3  ENT CHAR
```

where

N, M, O take on different values depending on the particular index characteristics, such as size of search key, extent of data field, and maximum number of entries per segment.

ID is a one byte code identifying all segments in the same index.

ANC is a pointer that indicates which entry in the segment can be used for the next index term to be stored. If the value of ANC is zero then the segment has the maximum number of entries or is full.

Each index segment must be initialized before operation as follows:

a. Each index field (KEE) is set to binary zero.

b. Each high sequence pointer is set to binary zero.

c. Each low sequence pointer is set with the subscript value of the next entry in the array (MINSTRUCT). LOW (N) is set to binary zero.

d. ANC is given the value (Ø1) (i.e., initially the first segment entry is to be used for the first index entry in the segment).

A generalized index search routine has been written which has multiple entry points depending on the number of discrete indices. (At present, eight indices are maintained; six corresponding to the six UOP levels, one for data definitions and one for keywords.) Upon access, this routine reads the first segment of the specified index, which contains the anchor or start of the index list. It begins comparing the passed search key against the KEE's (stored keys) in the segment. If this is the first entry in the index, then KEE (1) in segment one equals binary zero and no match is found. The no match occurs whenever the contents of KEE differ from the passed search key. If the passed key and the entry in KEE match, then the routine returns to the calling program and passes back the subscript value of the matching entry and disk address of the index segment, currently in core memory.

When a no match case arises, a check is made to determine if the passed key is less than or greater than the index entry (KEE) being compared. Then, either the low sequence pointer (LOW) or the high sequence pointer (HIGH) is used to determine the next entry against which a comparison is to be made. Before picking up this next entry the following checks have to be made on the pointer value:

a. If it is equal to binary zero, then we are at a terminal node, i.e., an entry for the passed key does not exist in the index. In this case a zero subscript value is returned that indicates no matching entry (terminal node) to which a link must now be made. The back pointer will be either a negative or a positive value, depending on whether the link should be made to the terminal node's low or high sequence pointer, currently in core memory. As usual, the disk address of the index segment is also returned.

b. If it is less than a threshold value, then the value is a subscript to an entry within the index segment, currently in core memory. The threshold is current set at 255 which is the maximum number of index entries permitted in a given segment. The search program uses the subscript to pick up the next entry and repeat the comparison operation.

c. If it is greater than the threshold, then the value has a double meaning; namely, it contains the disk address of the index segment in which the next entry can be found and the subscript value of that entry within the segment. The search program uses the disk to overwrite the current core resident segment with the new segment. The subscript value is then used to pick up the desired entry and repeat the comparison loop.

Thus, two returns from the search routine are possible, either a match or a no match. In the match case, the calling program performs whatever processing is required using the index entry and rewrites the index segment to disk upon completion, if the specified index entry has been modified. In the no-match case, either an error condition exists or the calling program wants to add a new index entry. In the former case, some appropriate error processing should be performed. In the latter case, i.e., index entry load, the calling program is responsible for finding an empty slot that can be used for the new index entry. To do this, the ANC Field of the index segment is used since it points to the next available slot in the segment.

If the ANC value for the index segment, currently in core memory, is non-zero, then the new entry can be inserted in the current segment and the ANC value is the subscript to this available space. Before using the indicated entry space, the calling program must replace the current ANC value by the value of the LOW pointer in the indicated entry space. Thus, for subsequent users, ANC will have an appropriate subscript value and continue to point to the next available entry. The new entry is initialized as required by the calling program and both LOW and HIGH pointers are set to zero, making the new entry a terminal node. The returned back pointer value is used to make the necessary linkage with the last compared key to preserve the logical ordering of the index.

When the value of ANC in the current core resident segment equals zero, then the current segment is full and cannot hold a new index entry. Since the LOW pointer of the last entry in each segment is initialized to zero, when this entry is used, ANC will pick up a zero value. In this case, empty space must be found in some other segment of the index. Segments of the index are retrieved sequentially until a segment is found whose ANC value is non-zero. Note, the starting point for segment retrieved is specified by a system parameter, like the anchor pointer, which gives the disk address of the first segment of the index which has empty space. As an index is initially built, this address will click up sequentially; however, whenever an index entry is deleted, thus, creating a hole in the index, this address will be reset to the segment from which the deletion was made. Thus, empty space will be reused.

VIII

DEBUGGING SUPPORT

1. Introduction. We have described in Volume 3 of this report a language designed to facilitate programming of conversational processes. This language, called the CAINT Executive Language (CEL) is employed in writing executive programs to control the logic of question selection and wording, response analysis, and various other activities. CEL-written programs require debugging, and ESDP is designed to include a support package specifically tailored to assist the executive programmer in debugging.

2. Debugging Support Capabilities. We define five basic debugging capabilities: (1) halt, (2) display, (3) alter, (4) change, and (5) begin. The capabilities will be presented by commands of the same name, and it will be possible to embed the commands in CEL.

For example,

```
IF (Condition ) THEN DISPLAY (Variable) ELSE HALT.
```

3. Commands.

a. HALT

Halt the CEL program and continue with the next sequential instruction when the start button on the user's console is pressed.

b. DISPLAY

Display the contents of named variables (in core or external storage).

For example,

```
A = 1; B = 6; C = A + B;

DISPLAY (C);
```

results in a printout of:

```
C = 7
```

c. ALTER

Enables the programmer to modify some area of core or external storage.

For example,

Programmer types:

C = A - B (continuing the above example)

C is set equal to -5.

d. CHANGE

Enables the programmer to change statements in the CEL program being debugged.

There are three forms of CHANGE defined:

(1) CHANGE (DELETE (statement number) TO (statement number))

(2) CHANGE (REPLACE (statement number) TO (statement number) WITH source code)

(3) CHANGE (INSERT AFTER (statement number) source code)

4. <u>Use of Debugging Capabilities</u>. Data value changes may be traced throughout the execution of a program.

For example,

```
DISPLAY (X);
```

means display the current value of X every time X changes value.

```
IF (Y > 5) & (Y <10) THEN DISPLAY (X);
```

means display the current value of X every time X changes only if Y is greater than 5 but less than 10. In other words, rather than inserting this statement in the CEL program every time that X is changed, the programmer can state the condition and desired action once, and the command will be in effect throughout the program execution.

Another option is deferral of printout.

For example,

```
DISPLAYD (X);
```

means record all value changes of X and print off-line.

A particular CAINT application might be a deferred display of all the question (in sequence) output for a given run.

During a debug run some means of data base protection would have to be provided. This might take the form of putting the data base in a read-only mode for each debug application.

This would mean that any area of the data base could be read, but writing would be directed to a scratch file, and any attempts to read "changed" data base records would also be directed to the scratch file (onto which they had previously been written).

A typical CHANGE and ALTER situation might occur following detection of a bug.

A CHANGE command (for replacement, deletion, or insertion of statements) could be used to attempt error correction. An ALTER command could then be used to reinitialize data variables to reasonable values for the point at which the program is restarted (using the BEGIN command).

5. __Methods of Implementation__. There are two general methods to support the CEL with the debugging capabilities described above: compilation and interpretation.

Some pre-processing would be required to prepare a deck for compilation so as to support the debugging capabilities described above. This could be coupled with an interrogation designed to elicit from the programmer the specific debugging requirements for each program.

Consider a simple example of the kind of preprocessing under discussion.

The programmer writes the following code in which the numbers on the left are machine generated statement numbers, to be used by the programmer as operands of a CHANGE command.

```
S00100    L1:  DO;
S00200         IF UOP.NUMENS = Ø THEN CALL OUT ('NO MEMBERS',
S00400            UOPAD); ELSE CALL OUT (MEMLIST,NUMENS);
S00500         IF UOPAD = UOPEND THEN GO TO L2;
S00700         UOPAD = UOPAD + 1;
S00800         CALL NEXT (UOPAD);
S00900         GO TO L1;
S01000    L2: END
```

The following is an example of a dialogue that could then take place:

```
MSG  Type the number of each command to be used

                         1.  HALT

                         2.  DISPLAY

                         3.  ALTER

RES  2

MSG  Which variables are to be displayed?
```

RES UOP.NUMENS

MSG Which variables are to control the display of UOP.NUMENS?

RES UOP.NUMENS

MSG For which new values of UOP.NUMENS is UOP.NUMENS to be displayed?

RES UOP.NUMENS > Ø

MSG Pre-processing has begun

MSG Compilation has begun

MSG At which statement do you want execution to begin? (Type Number 1 - 10).

RES 1

MSG Execution has begun

Etc.

The deck produced by the pre-processor looks as follows:

```
L1L2:  PROC OPTIONS (MAIN);

%INCLUDE  DATA (DCL1);   Includes declarations necessary to define
                          data base references in program.

%INCLUDE TEMP (CODE);   Includes code to initialize variables in
                          support of DISPLAY.  (This code was
                          generated as a result of the dialogue
                          pictured above.)

%INCLUDE  PROG  (CHECK);   Includes this system (ESDP) program as
                          internal procedure.  This program
                          supports the debugging capabilities
                          described above.

LAB1(1):L1:DO;           LAB1(1) defines this statement as an
                          element in the label array LAB1(n)
                          (where N = 10 in this case).  Each
                          statement in the original will be
                          prefixed in this way to allow
                          implementation of the BEGIN command.

LAB1(2):IF UOP.NUMENS = Ø

         THEN

LAB1(3):  CALL OUT (10 MEMBERS,
LAB1(4):  ELSE CALL OUT (MEMLIST,

                   NUMENS);

                         CALL CHECK:

LAB1(5):  IF UOPAD = UOPEND

          THEN

LAB1(6):  GO TO L2;

LAB1(7):  UOPAD = UOPAD + 1;

                         CALL CHECK:

LAB(8):  CALL NEXT (UOPAD);

                         CALL CHECK;

LAB1(9):  GO TO L1;

LAB1(10):  L2:  END;
```

In summary:

The HALT command could have been enabled by the same interrgation process which enabled the DISPLAY command. The program can still be halted by the programmer by pressing the stop button on the console.

The CHANGE command can be utilized following a halt by means of a dialogue with the CHECK routine.

The DISPLAY command has been selectively enabled through interrogation.

The ALTER command could have been enabled by means of interrogation.

The BEGIN command is supported by embedding the program in a label array as shown.

The main differences between compilation and interpretation are (1) interpretation will effect some implementation costs whereas the compiler is essentially free, (2) interpretive program CHANGES can be made more rapidly, because it is not necessary to recompile and linkage edit, and (3) interpretive debugging commands can be entered at execution time.

IX

BIBLIOGRAPHY

[1] ________ *System/360 Flowchart (360A-SE-22X) User's Manual*, H20-0293, IBM Corporation, White Plains, N.Y.

[2] ________ *IBM 1440/1460 Administrative Terminal System Application Description*, Form H20-0129, IBM Corporation, White Plains, N.Y.

[3] Mills, H. D. and Michael Dyer, Evolutionary System for Data Processing, *Proceedings of IBM Real Time Systems Seminar*, Houston, Texas, November 1966.

[4] ________ *IBM 7090/94 Autoflow System User's and Operator's Manual*, Applied Data Research, Inc., Washington, D. C., 1967, under NASA Contract No. NAS5-10021.

[5] ________ *IBM Operating System/360 Concepts and Facilities*, Form C28-6535, IBM Corporation, White Plains, N.Y.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Center for Exploratory Studies, International Business Machines Corporation, Rockville, Maryland 20850	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP N/A

3. REPORT TITLE
EVOLUTIONARY SYSTEM FOR DATA PROCESSING PROGRAMMING SPECIFICATIONS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
None

5. AUTHOR(S) *(First name, middle initial, last name)*
Charles T. Meadow
Douglas W. Waugh
Gerald F. Conklin
Forrest E. Miller

6. REPORT DATE January 1968	7a. TOTAL NO. OF PAGES 47	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO. F19628-67-C-0254	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-68-143, Vol. IV	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	
d.		

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Command Systems Division, Electronic Systems Division, Air Force Systems Command, USAF, L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT
ESDP is a proposed system whose purpose is to acquire, store, retrieve, publish and disseminate all documentation, exclusive of graphics, concerned with a large computer programming activity. Documentation is deemed to consist, not only of final or formally published after-the-fact reports, but of working files, design and change notices, informal drafts, management reports--in fact, the entire recordable rationale underlying a programming system. Maximum attention has been concentrated on the means of acquiring and organizing documentation. Two major, complementary approaches are proposed. The first is called Program Analysis and is a process of extracting documentation directly from completed programs. The second is called Computer Assisted Interrogation and is a process of eliciting information directly from the programmer, through on-line communication terminals. The former provides canonical data about the program's structure. The latter provides explanatory material about all aspects of the program, and in the absence of canonical data, may provide tentative structural information as well. The conclusion of the study group is that ESDP is a feasible concept with present-day technology and that it will materially benefit using organizations in the production of programs and in guiding their evolution as requirements change. Its value will be greater for larger organizations, whose internal communications difficulties tend to cause truly gigantic inefficiencies. Its implementation as a support system for such projects would require a significant quantum of investment in order to produce these benefits and is predicated on the use of a computer system dedicated solely to the use of ESDP.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification