April 1971

A Command and Query Language Assembler for an Extended Data Management System

Jorge Gana
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-71-22.

A Command and Query Language Assembler for an Extended Data Management System

Abstract

For a data management system with information storage and retrieval capabilities a language is needed by which a user of the system can specify the records he wishes to retrieve and the operations he wishes to perform on these records. The Command and Query Language under discussion was developed to meet these needs for the extended data management system. Its development was divided into two spheres of responsibility. The first sphere, referred to as the Assembler, centers on the routines needed for accepting and translating user requests. The second sphere centers on those routines needed for executing the translated requests. These routines are called collectively the Interpreter. The design of the Command and Query Language and the implementation of the Assembler is the topic of this report.

Basically, the Language enables the user to specify the records by means of the logical and arithmetic expression of keywords. Since program names may be keywords, the user can specify operations (to be performed on records) with keyword expressions as well The design of the Language involves the following steps:

(1) Define the requirements of the Language.

(2) Define the (external) syntax and semantics of the Language.

(3) Design an internal form of the Language to allow efficient processing by the Interpreter.

The design and implementation of the Assembler will result in the necessary routines which can check the syntax of the Language and transform the Language from its external syntax to internal form.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-71-22.

University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
Philadelphia, Pennsylvania

TECHNICAL REPORT

A COMMAND AND QUERY LANGUAGE ASSEMBLER FOR AN EXTENDED DATA MANAGEMENT SYSTEM

by

Jorge Gana

April 1971

Submitted to the
Office of Naval Research
Information Systems Branch
Arlington, Virginia

under
Contract N00014-67-A-0216-0014
Research Project NR 049-153

Moore School Report No. 71-22

A COMMAND AND QUERY LANGUAGE ASSEMBLER FOR AN EXTENDED DATA MANAGEMENT SYSTEM

Abstract

For a data management system with information storage and retrieval capabilities a language is needed by which a user of the system can specify the records he wishes to retrieve and the operations he wishes to perform on these records. The Command and Query Language under discussion was developed to meet these needs for the extended data management system. Its development was divided into two spheres of responsibility. The first sphere, referred to as the Assembler, centers on the routines needed for accepting and translating user requests. The second sphere centers on those routines needed for executing the translated requests. These routines are called collectively the Interpreter. The design of the Command and Query Language and the implementation of the Assembler is the topic of this report.

Basically, the Language enables the user to specify the records by means of the logical and arithmetic expression of keywords. Since program names may be keywords, the user can specify operations (to be performed on records) with keyword expressions as well

The design of the Language involves the following steps:

(1) Define the requirements of the Language.
(2) Define the (external) syntax and semantics of the Language.
(3) Design an internal form of the Language to allow efficient processing by the Interpreter.

The design and implementation of the Assembler will result in the necessary routines which can check the syntax of the Language and transfrom the Language from its external syntax to internal form.

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) University of Pennsylvania The Moore School of Electrical Engineering Philadelphia, Pa. 19104	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

A COMMAND AND QUERY LANGUAGE ASSEMBLER FOR AN EXTENDED DATA MANAGEMENT SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report

5. AUTHOR(S) (First name, middle initial, last name)

Jorge Gana

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1971	70	8

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-67-A-0216-0014 b. PROJECT NO. NR 049-153 c. d.	Moore School Report No. 71-22 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Office of Naval Research Information Systems Branch Arlington, Virginia

13. ABSTRACT

For a data management system with information storage and retrieval capabilities a language is needed by which a user of the system can specify the records he wishes to retrieve and the operations he wishes to perform on these records. The Command and Query Language under discussion was developed to meet these needs for the extended data management system. Its development was divided into two spheres of responsibility. The first sphere, referred to as the Assembler, centers on the routines needed for accepting and translating user requests. The second sphere centers on those routines needed for executing the translated requests. These routines are called collectively the Interpreter. The design of the Command and Query Language and the implementation of the Assembler is the topic of this report.

Basically, the Language enables the user to specify the records by means of the logical and arithmetic expression of keywords. Since program names may be keywords, the user can specify operations (to be performed on records) with keyword expressions as well.

The design of the Language involves the following steps:

(1) Define the requirements of the Language.
(2) Define the (external) syntax and semantics of the Language.
(3) Design an internal form of the Language to allow efficient processing by the Interpreter.

The design and implementation of the Assembler will result in the necessary routines which can check the syntax of the Language and transform the Language from its external syntax to internal form.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Assembler						
COBOL						
Command language						
Control programs						
Data management						
FORTRAN						
Interfacing						
Interpreter						
Logic						
Query language						
Semantics						
Syntax						
Systems design						
Translation routine						

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER 1

INTRODUCTION

In information storage and retrieval a language is needed by which a user of the facility can specify the records he wishes to retrieve, and the operations he wishes to perform on those records. We called the language "Query Language" [1]. The Query Language for an Extended Data Management System should not be restricted to any standard information retrieval language, (e.g., languages suitable for document retrieval only) particularly since it is going to be used by a large number of users of different disciplines who need to share the resources and data of the system in a common storage. The Query Language should be capable of providing flexibility in all levels, e.g., enabling the user to bring a program from his file for execution and providing this program with parameters or data which in turn can be new executable programs or records. For specifying records, the language should allow the user to describe records in terms of the logical and arithmetic expression of keywords. These records may be data records or programs. They are both stored in files. Records may be processed by system programs or by user's programs. The former is accomplished through the use of commands, and the latter through the use of the query language.

The subject of this report is the design and implementation of a Command and Query Language in agreement with the requirements just described above, which can be summarized as follows:

1) The user must be able to write procedures, have them executed, and, if desired, have them stored for repeated use at a later time.

2) The user must be able to request the execution of any of his own programs stored in his files, any pre-defined system routines, any

procedures he has defined, and any programs or procedures of other users to which he has access rights.

3) The user must be able to supply input parameters and data to programs or procedures he is using, providing format information about these parameters or data where necessary.

4) The user must be able to provide logical expressions of key-words as data, and provide abbreviations (called local names) for long expressions to facilitate procedure writing.

The Command and Query Language under discussion was developed to meet these needs for an extended data management system.

The language format was designed with the following requirements in mind:

1) The format must allow the writing of statements to satisfy all the above requirements.

2) The format should be as simple as possible to facilitate the ease of using the language.

3) The format should not require writing excessive amount of material unless the material aids in the user's understanding.

Its development was divided into two spheres of responsibility. The first sphere, referred to as the Assembler, centers on the routines needed for accepting and translating user requests from external to internal encoded format. The second sphere centers on those routines needed for executing the translated request. These routines are called collectively the Interpreter.

The design of the Command and Query Language involves the following steps:

1) Define the requirements of the Language.

2) Define the (external) syntax and semantics of the Language.

3) Design an internal form of the Language to allow an efficient processing by the Interpreter.

The design and implementation of the Assembler will result in the necessary routines which can check the syntax of the Language and transform the Language from its external syntax to its internal form.

CHAPTER 2

OVERALL SYSTEM DESIGN CONSIDERATIONS

In order to define the requirements of the query language, it is necessary first to specify the objectives and design goal of the Extended Data Management System [2]. Many suggestions were made at the beginning of the design development of the system as to ways the system should be implemented, areas of useful research or things the system should do. The more important suggestions, that have relevance in relation to the query language design, together with the result of the consideration of these suggestions, are presented below.

2.1 Interfacing with the Existing Operating System

The RCA Spectra 70/46, on which the Extended Data Management System is to operate, has a powerful general purpose, time-sharing operating system (TSOS). In order to make available to the users all our new facilities, it was decided to build our system as a part of the existing TSOS system and to make use of its facilities as much as possible. This adds the flexibility that all facilities are given to the user as part of the general purpose system instead of a dedicated system. This decision involved three major areas of the operating system:

1. Use of Existing Control Program Elements
2. Use of Existing Data Management System (DMS)
3. Use of Existing Command Language Facilities

The Command Language for RCA TSOS not only has extensive features in its own right, but contains certain features which not only make for flexible use of the Command Language, but are helpful in adding new commands to the system. One of these features, is the Interpretive Scan Processor (ISP), a table driven interpreter which allows new commands

to be defined using existing system macro-instructions. Another is the Terminal Command Processor (TCP), which decodes the commands as they come in, calls the ISP to interpret them, then gives control to the appropriate servicing routine to handle the command. It was decided that this existing Command Language system, with certain modifications would be completely suitable for handling our command initial processing requirements.

2.2 Use of High-Level Source Programming Language

Another consideration was that of using a high-level source language (such as FORTRAN or COBOL) for programming the system, rather than assembly language - the traditional source language for system programming.

The obstacle which forced the project to abandon this idea was that our system routines, like most of the rest of the TSOS components, have to be reentrant, i.e. usable by several programs at the same time. Special programming techniques must be used to produce reentrant programs, and it was found that the object programs produced by the high-level programming languages available under TSOS could not be made reentrant. The possibility of converting a high-level language compiler from another computer (e.g. IBM 360 PL/1) which could produce reentrant object programs was investigated, but it was found that the conversion problems were too great for this idea to be used. It was therefore decided to write our system routines in assembly language, using reentrant coding techniques.

2.3 TSOS Compatibility

One of the design goals of the Extended Data Management Facility was to have a non "dedicated" system to our requirements, but rather a

general purpose time-sharing facility. By general purpose we mean that it can be used by people who may use our special features, people who may use only the existing system features and people who may want to use both features. Thus it was decided that:

1) users who did not wish to make use of our advanced data management features should be able to run standard TSOS programs on the new system without change; and

2) the use of our components by some users should not degrade performance for other users of the system.

These considerations affected the decision on whether or not to use the existing Data Management System and how to modify the Command Language.

CHAPTER 3

THE COMMAND AND QUERY LANGUAGE

The development of the Command and Query Language and its peripheral requirements was divided into two distinct spheres of responsibility. The first of these spheres centers on routines referred to as the Assembler needed for accepting and translating user request. The basic assembly routine (STATRAN), the logical expression translation routine (LOGTRAN) and other routines of the Assembler are discussed in Chapter 4 of this thesis. The second sphere centers on those routines needed for transforming the translated requests into a form which the Supervisor of the Extended Data Management Facility will accept and act upon. These routines are called collectively the Interpreter [3]. Based upon the user requests, the first routine of the Interpreter sets up a modified TSOS Command Language Procedure File which is the standard TSOS convention for placing the incoming terminal commands and statements as briefly discussed in Section 3.1. The Command Language Procedure File can then be processed by other routines of the Interpreter for the appropriate actions as requested by the user. Following is a brief discussion of some Interpreter functions followed by the complete Assembler requirements in detail.

3.1 The Interpreter

The following is a brief discussion of some functions of the RCA time sharing operating system which is essential in understanding the mechanism of the Interpreter and its interface with the Assembler.

3.1.1 TSOS Procedure (PROC) Files

The RCA Time Sharing Operating System reads all commands from a system logical file named SYSCMD. If the tasks are non-conversational (i.e., entered from the card reader), SYSCMD is defined as the card reader (or more specifically the SPOOL file created from the card input). For conversational tasks (entered from terminals), SYSCMD is defined as the terminal the user is on. In this case, the user may directly issue system commands from his terminal. TSOS also provides a method whereby SYSCMD may be redirected so that the system accepts commands from a temporary command file which has been cataloged in the system. Such a temporary command file, created in the same way as any other file, is referred to as a Procedure File or PROC File. Much of the detailed information concerning PROC files is not pertinent in this thesis, and may be obtained from the appropriate RCA publications. However, since certain information about PROC files is considered critical and essential to the understanding of how these PROC Files are used by the Assembler and Interpreter, this material is covered below.

3.1.2 PROC File Commands

A TSOS PROC File must be a cataloged sequential or indexed-sequential file. It must contain nothing but defined TSOS Command Language statements. In particular, the PROC File must begin with the /PROCEDURE (or /PROC) command and end with the /ENDP command. The command /DO [proc file name] will direct the system to begin accepting commands from the specified PROC file instead of the current command input file, i.e. the SYSCMD. A PROC File can contain DO commands, and the PROC file thus referenced can also contain DO commands to any level. However, a PROC File cannot return control to the PROC File that called it. Control

is always returned to the original command file (reader or terminal). The /ENDP command is the command used to return control from a PROC File to the primary command input file. There may be several /ENDP commands in a PROC File. When it is encountered, anyone of these /ENDP commands will return control to the primary command file.

3.1.3 Use of TSOS PROC Files

A typical use of the standard TSOS PROC File can be illustrated as follows: the job of compiling, linkage editing and executing a source program under TSOS requires approximately 12 Command language statements, most of which are the same for any job of the same type. The PROC File allows these common statements to be placed in a file once, to be called by a programmer for repeated use without having to write out all of the control statements each time they are used.

3.1.4 Terminal Command Processor

The existing TSOS Terminal Command Processor and certain of its commands have shown to be useful to our implementation. Through the use of these commands, the Terminal Command Processor is incorporated to do a large share of the work assigned to the Interpreter.

To understand how the Terminal Command Processor is utilized we should first briefly discuss how it works and also one special capability that is associated with it.

When a user issues a command from a terminal, the Terminal Command Processor reads this command and then scans a table consisting of the names of each TSOS command and the address of the command servicing routine for handling that particular command. It then branches to that routine. The special capability mentioned above is that a user may create a PROC File consisting of a list of TSOS commands, catalog and

store it, and then have the Terminal Command Processor process the list of commands at some later time. Processing of the list is initiated by issuing the command /DO [file name], where file name is the name of the user created file containing the list of commands. The last command of the list should be the /ENDP command. The routine associated with this command is a "clean up" routine. It closes the PROC File, releases any memory acquired by the /DO command, and resets switches to allow the Terminal Command Processor to return to normal processing methods, i.e., directing the system to accept commands from the SYSCMD again. The above described functions were used extensively by our Query Language processors. The concept of procedure is expanded in our facility to include any set of storage and retrieval statements and new commands in addition to the TSOS commands. Furthermore, the use of PROC Files makes possible the use of formal parameters and local names in sets of retrieval requests.

Since the addition of new commands and their respective servicing routines can be done without too much difficulty to the existing Operating System, and since this would be a good way to make certain functions of the Data Management System quickly and easily available to the user, the decision was made to add certain new system functions as commands to the Command Language. These commands include a formal parameter command, a retrieve command, an execute command, and a continue command. For detailed information about these commands and further Interpreter functions see [3].

3.1.5 Interface With the Assembler and Interpretation Flow

The flow of events following the initiation of the assembly processes will now be discussed. The Assembler itself is run as a user

program (a Class I program). This allows direct control by the user over the assembly process, and enhances future modifications of the Assembler and the use of the function BREAK to interrupt and resume the assembly process.

Initially, after it takes control and before it accepts any data from a terminal, the Assembler passes control to the Interpreter. At this point the Interpreter catalogs, allocates, and opens a file to be used as the PROC File. The name given to this file is the one specified by the user as the name for his procedure. Also the command /PROC is placed into the file as the first statement (sometimes called 'record') of the file. Finally control passes back to the Assembler.

Whenever the Assembler accepts from the user an executable statement, control is passed to the Interpreter again. The Interpreter now does one of three things before passing control back to the Assembler.

1. If the statement was a TSOS command then the command is entered as a record in the PROC File exactly as it came from the terminal;

2. If the statement was a request for execution of a user program, then the command /EXEC [program name] is entered as the record in the PROC File. This command is an existing TSOS command when the user program doesn't have any parameter. If the user requests the execution of a program which requires parameters (e.g., the program may require a logical expression or other data as parameters), then the command

/MEXEC,[program name],PARAM=C'[parameter]'

is entered as the new record in the PROC File. /MEXEC is a new command. It is added for the introduction of parameters to a program. At the moment this new command accepts only one parameter. Considerations are being given to implement it with the capability of accepting more than

one parameter. Eventually, the command /MEXEC will replace the present /EXEC command of the Time Sharing Operating System.

3. If the statement was a request for the execution of a previously written procedure (another stored PROC File), then that PROC File is retrieved and each of its records (commands) is entered in the new PROC File.

Finally, when the Assembler reads an end statement, control is again passed to the Interpreter. This time it adds the command /ENDP to the PROC File, closes the file, and then passes control back to the Assembler.

Now, all the user has to do to execute the procedure that he has written is to issue the command /DO [file name].

What has been discussed so far is basically the working of the Interpreter and its interface with the Assembler as in Figure 1.

A capability that was added into the system was to include a parameter in the /PROC command (first command of a PROC File). The command is in the form: /PROC C. Our previous description of PROC Files is not changed by the addition of this new parameter 'C', whose only effect is, at execution time, to point to a list of the actual records stored in the PROC File being executed. Its use will be illustrated in the later sections.

3.2 The Assembler

In order to translate a user request into a convenient internal format for processing by the Interpreter, an Assembler was designed which performs the following functions:

1. It checks the external syntax of the incoming requests, and provides the user with appropriate guidance in sending correct

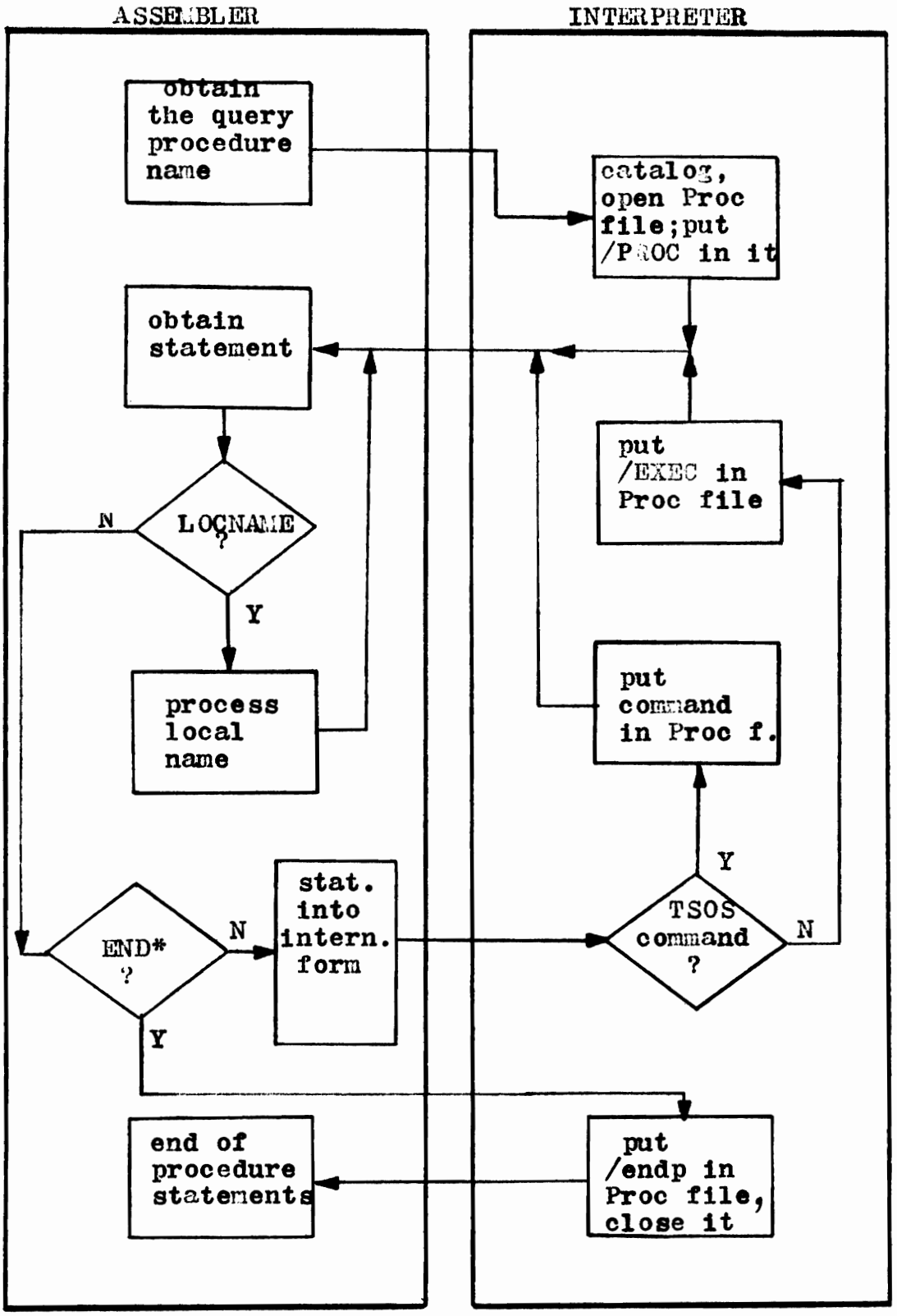

ASSEMBLER
INTERPRETER
obtain the query procedure name
catalog, open Proc file;put /PROC in it
obtain statement
put /EXEC in Proc file
N
LOCNAME ?
Y
process local name
put command in Proc f.
Y
stat. into intern. form
END* ?
N
TSOS command ?
N
Y
end of procedure statements
put /endp in Proc file, close it

Figure 1

requests.

2. It accepts data from the user's procedure and constructs a record in special form which is used by the Interpreter in processing the procedure.

3. It translates logical expressions of keywords into a variable length internal record which contains all the necessary information required by the Request Supervisory Subsystem [4], [5] for processing the specific request.

We now describe the Query Language and its external syntax. We will then describe the Assembler.

3.2.1 Syntax and Semantics

Normally the user communicates with a Data Management Facility by writing procedures.

A procedure should contain the following elements, some of which are optional, others required.

1) Procedure Name - The procedure name allows the procedure to be retrieved by the user for later use. This requirement was also present in the Operating System, since a name is associated with the creation of a PROC File and in order to execute it at a later time the user should issue the /DO [procedure name] command.

2) Formal Parameters - The formal parameters act as "dummy" variables in the procedure and are given values when the procedure is actually used. The use of formal parameters should be optional. This option was not available in the operating system. By its nature, however, it was considered an option to be handled by the Interpreter rather than by the Assembler, because substitution of parameter values is done at the time of procedure interpretation. At first, the idea of

incorporating this function into the DO command was considered. However it was felt that it was no more difficult for the user to learn and use the new /FPARAM command than to learn and use a modified version of the existing /DO command. From the implementation point of view the construction of a separate formal parameter command was much better than modifying the existing /DO command.

The format for this command is given in 4.3.1.

3) Local Names - Local names are used to refer to or to abbreviate long descriptions which are used frequently in a procedure. The use of local names should be optional.

This facility was also not available in the present Operating System and it was implemented as part of the program statements Translator Routine (STATRAN) of the Assembler, whose description is included in Chapter 4.

4) Statements - Statements are the most important elements of the procedure. One or more statements must appear in any procedure. A statement may have three sub-elements: a label (optional) to identify it, an operation (required) to name the program, system routine or stored procedure to be executed, and any input parameters or data needed by the operation during execution.

This last requirement can be also realized in the new system. This is done by means of the PROC file mechanism of the Command Language as described before. The Assembler enables the new data management system to accept user programs and parameters by placing the following new command in a PROC file:

/MEXEC[program name],PARAM=C'[parameter]

The Assembler was implemented in the same way as a Class I program. By issuing the /EXEC PROC command, the user can bring in the Assembler which in turn causes the STATRAN to process user's statements and requests.

Types of statements accepted by the STATRAN are:

1) regular TSOS commands

2) new commands for the Extended Facility, and

3) user programs.

However, all these statements will have exactly the same syntax, namely TSOS command language syntax. This relieves the user from learning another syntactical rule in using the new commands for the Facility.

An example of a sequence of commands in a procedure which will load a user program TEST1, stop it at locations X'10E' and X'1A6', execute another user program TEST2 and then the TSOS "desk calculator" command CALC is the following:

```
*LOAD TEST1
*AT L'10E'
*AT L'1A6'
*RESUME
*TEST2
*CALC
```

where the symbol '*' is printed by the system and indicates that the Assembler of the Facility is accepting data.

In the example RESUME and AT are regular TSOS commands of the Interactive Debugging Aid facility.

Furthermore, STATRAN accepts two more special Assembler commands:

1) LOCNAME command which indicates to the Assembler the presence of local names (synonyms to abbreviate long expressions) in the procedure and

2) END* command which terminates a normal procedure.

The LOCNAME command should be followed by 1 or more (up to ten) synonyms followed by the actual value the user wants to be replaced for it in the procedure. Each synonym should be prefixed by an '#' symbol and the whole LOCNAME statement is ended by three consecutive '#' symbols.

Before entering any statements to the procedure, a procedure name should be given. This procedure name is assigned by the user when he began the creation of the procedure by issuing the /EXEC PROC command.

An example of this initialization might be:

```
/EXEC PROC

  ENTER PROCEDURE NAME

*SEARCH
  BEGIN
*
```

After receiving the message from the Assembler, the user assigns the name SEARCH to his procedure. The system responds with the message Begin and the user can now input statements, one at a time.

Execution can be performed at a later time by issuing /DO SEARCH.

The following is a description of allowable Local Name statements in Backus Normal Form:

< LOCAL NAMES STATEMENT > ::= LOCNAME < SPACE > < PARAM > ###

< PARAM > ::= # < SYN > < SPACE > := < VALUE > |

< PARAM > < SPACE > < PARAM > | < PARAM > < SPACE >

< SPACE > ::= ⊅ | ⊅ < SPACE >

< SYN > ::= any alphanumeric string

< VALUE > ::= any expression the user wants to substitute by < SYN >

The following example might be such a local name statement:

```
*
*
*TEST1
*LOCNAME #A :=C'AUTHOR=SMITH&YEAR=1968'
*RETRIEVE $HORTON, #A,1,
*END*

     END OF PROCEDURE

/
```

The #A in the statement beginning with 'RETRIEVE' would be replaced by C'AUTHOR=SMITH&YEAR=1967' before control is passed to the Interpreter. Effectively these statements are stored as records in a PROC file for interpretation as follows:

```
/PROC C
/ ...
/ ...
/EXEC TEST1
/RETRIEVE $HORTON,C'AUTHOR=SMITH&YEAR=1968',1,
/ENDP
```

Upon receiving the END* Assembler command, the Assembler sends the message END OF PROCEDURE to the user and passes control to the Interpreter, which in turn closes the PROC file and the system returns to the command mode which is indicated by the slash '/' character printed at the terminal. From now on, the user can execute his procedure by issuing the /DO [procedure name] command.

Examples of a typical procedure creation, and the way they are called by the user can be seen in Appendix D.

3.2.2 Additional Requirements of the Assembler

So far we have not discussed how the user is going to use the language in order to retrieve certain specific records and how he is going to describe those specific records he wishes to retrieve and also the operations he wishes to perform on those records, particularly when the user doesn't want to write a complete procedure, but instead to communicate with the Facility by what may be a simple statement. This requirement adds flexibility to the system and since a method for adding new commands [6] was developed and was found to be relatively easy, it was decided that those functions which are a part of the Extended Data Management System should be quickly and easily available to the user as commands. In particular the Retrieve Command, Formal Parameter Command, Continue Command, and a new Execute Command were added as part of the Interpreter implementation and a Restore Command was added as part of the Assembler implementation since it requires more language processing. This Restore Command allows the user to make addition or deletion of specific lines in a particular record and restore it to the file. Complete description of the format of these commands, plus the RESTORE command in detail can be found in 4.3.

As has been explained before, logical expressions of keywords are very important in processing information storage on retrieval requests. Wherever such an expression appears as a parameter (i.e. in the RETRIEVE command) or input datum, a special routine is called in to translate the expression into an internal form. Any logical expression must be translatable into disjunctive normal form with at least one non-negated term

within each disjunct to prevent excessive file searching.

The allowable logical expressions can be described in Backus Normal Form as follows:

< ATTRIBUTE > ::= any key attribute

< VALUE > ::= any key value

< MULTIPLE VALUE > ::= < VALUE > | < VALUE > 'TO' < VALUE >

< RELATION > ::= 'LT' | 'GT' | 'LE' | 'GE'

< SIMPLE PRIMARY TERM > ::= < ATTRIBUTE > 'EQ' < MULTIPLE VALUE > |
< SIMPLE PRIMARY TERM > 'OR' < MULTIPLE VALUE >

< PRIMARY TERM > ::= < ATTRIBUTE > < RELATION > < VALUE > |
< SIMPLE PRIMARY TERM >

< SIMPLE TERM > ::= < ATTRIBUTE > 'NE' < MULTIPLE VALUE > |
< SIMPLE TERM > 'OR' < MULTIPLE VALUE >

< TERM > ::= < PRIMARY TERM > | < SIMPLE TERM >

< DISJUNCT > ::= < PRIMARY TERM > | < DISJUNCT > 'AND' < TERM >

< LOGICAL EXPRESSION > ::= < DISJUNCT > | < LOGICAL EXPRESSION > 'OR'
< DISJUNCT >

3.2.3 Service Request Options

The Request Supervisor Subsystem [4], [5] provides the user with many service request options. This has been done in order that he can use these facilities to manipulate his files as he desires, particularly when he enters the Facility in a non-conversational mode, e.g., by cards instead of from a terminal. Each request is in the form of a macro. Each macro instruction generates a group of assembly language statements. One of the statements generated is a supervisor call. The supervisor call instruction (SVC) enables the machine to switch from any state to the Interrupt Control State (P_3), i.e. the SVC causes an interrupt. In

this state, and through the use of the Interrupt Analyzer, the Subsystem determines what to do. Statements that accompany the SVC in a macro expansion supply the necessary parameters for processing the user's request. Once the system knows how to respond to the particular interrupt, it switches to state P_2, where interrupt responses are handled.

Macro instructions are extremely useful since they are located in a macro library accessible to all users. It was therefore decided to incorporate the Logical Expressions Translating routine (LOGTRAN) as a macro instruction. This will add the flexibility that not only the system programs (e.g., the RETRIEVE command) can call upon it in order to process their parameters consisting of logical expression of keywords, but also the user program can call upon it if the program contains logical expressions as parameters.

All the user needs to know in order to use it, is the proper way of calling the macro; all the other steps, the generation of the assembly language instructions and the SVC itself are done internally by the Assembler.

3.2.4 Adding and Testing New Supervisor Calls

To be classified as an operating system component, a program must be linked to the operating system before the system is loaded into the computer. This linking process is normally very time consuming. It involves up to six hours of computer time in the case of TSOS with slow tape drives.

Obviously it would be prohibitive in terms of both computer and programming time if each time a new routine must go through the above linking process to be tested with the system. A better way was found by which a special modified version of TSOS can be generated which allows

new routines to be added to the system and run as system components very easily [6]. This, in particular, allows easier addition of SVC servicing routines. This is the way by which the Logical Expressions Translating routine (LOGTRAN) was tested and implemented.

CHAPTER 4

DESIGN AND IMPLEMENTATION OF THE ASSEMBLER

In order to accomplish the necessary language translation into an internal format for processing the specific request, the Assembler comprises the following programs:

a. Logical expressions translation routine (LOGTRAN) - The purpose of this program is to translate into its internal form a Logical Expression appearing in a statement given by the user to the Extended Data Management Facility through the query language. Basically, the internal form is composed of a Key Information Buffer and a Description Control Block. At execution time of the statement the block is used by the retrieval routines of the Supervisor to perform the required search based on the attribute and value pairs in the block.

b. Statement translation routine (STATRAN) - This routine will prepare and setup the necessary information about the user's statement and pass it to the Interpreter. It will also process all Local Names, if they are present in a statement, before passing control to the Interpreter.

4.1 Logical Expressions Translation Routine (LOGTRAN)

At present this program accepts as its input a Logical Expression in "external" Disjunctive Normal Form (EDNF). Blanks are ignored provided they do not appear inside a keyword. For example, if we have a logical expression as follows

John Smith 'OR' ..

then the blank space between John and Smith is stored as a blank character while the blanks between Smith and 'OR' are ignored. This enables the user to use blank characters as part of the keywords.

The permitted relations and their user codes are as follows:

RELATION	USER CODES
EQUAL	'EQ', =
NOT EQUAL	'NE'
LESS THAN	'LT', <
LESS OR EQUAL THAN	'LE'
GREATER THAN	'GT', >
GREATER OR EQUAL THAN	'GE'

and the codes for the logical connections are:

'AND', &

'OR', V

'TO'

If a keyword contains the relation NOT EQUAL, then this description will be satisfied by any record with the same attribute and a different value. Note that this does not exclude records that have the same attribute and value as keyword.

Considerations are given to the design of the internal format specifications of a logical expression. These are:

1. As much control information as possible should be removed from the logical expression itself and stored in a separate block, the Description Control Block, allowing only this block to be passed to the Supervisor with pointers to the information required for a particular retrieval. This arrangement permits the later information, the Key Information Buffer, to be in a completely variable length format.

2. The control information in the Description Control Block should be easily handled by the Interpreter.

Thus, the LOGTRAN program produces as its output a Key Information Buffer and a Description Control Block. The Buffer and Block constitute the internal representation of an external disjunctive normal form received by the LOGTRAN.

Key Information Buffer (KIB):

This area contains the actual attribute and value pairs as given by the user without any truncation. There are no delimiters or control characters stored in this buffer. For example, the keywords

AUTHOR = SMITH 'AND' YEAR = 1940

will reside in the Key Information Buffer in the following way:

AUTHORSMITHYEAR1940

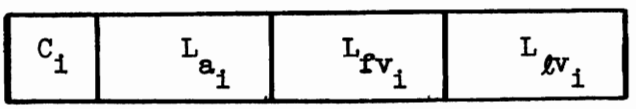
The information needed to accomplish the necessary retrieval and to sort out the keywords is stored in the Description Control Block (DCB).

Description Control Block (DCB):

It has two kinds of fixed length elements. The first one contains three sections: 1) a relative address section, 2) a control code section, and 3) format number pointers.

4 bytes 10 bytes 4 bytes

The control code section contains the following codes:



1 byte 3 bytes 3 bytes 3 bytes

where C_i = code to indicate RELATION between attribute and value

L_{a_i} = length of i^{th} attribute

L_{fv_i} = length of i^{th} value (or of first value of the i^{th} keyword if a case of multiple values is implied)

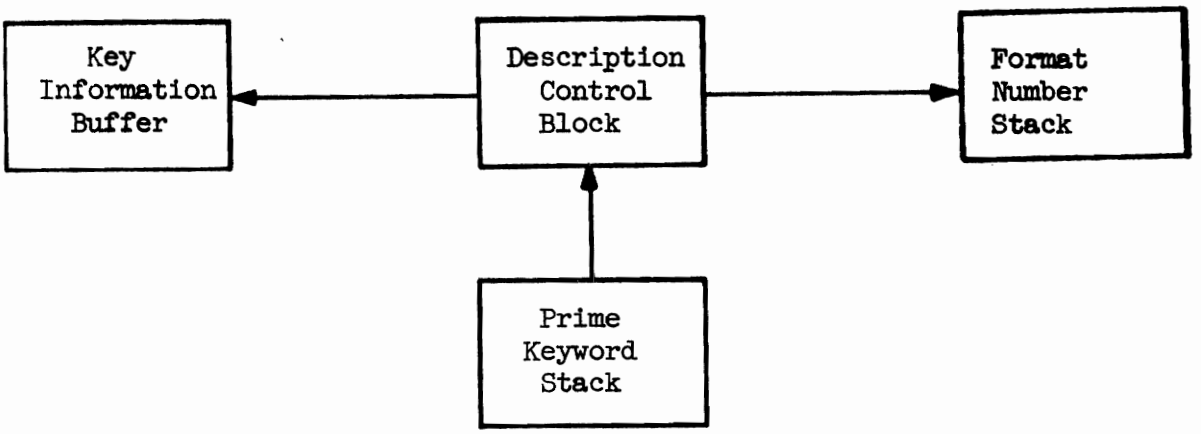
$L_{\ell v_i}$ = length of last value if a case of multiple values is implied. If not, it is left blank.

The third section is for the pointers to format numbers. It is used to help the record checking routine.

The counter tells how many format numbers there are in a list of format numbers which are associated with that particular keyword.

The second element in the DCB is of the same length of the previous element. It is left empty with exception of the first byte, where a control code is set there to indicate the end of an elementary conjunct to the search routines.

Both Key Information Buffer and the Description Control Block are used by the Supervisor along with its Prime Key Stack. Those blocks and stacks constitute the basic control information which enables the Supervisor to direct the primitive storage, retrieval and dissemination routines of the Storage and Retrieval Subsystem for honoring the user's request. The interconnection of these blocks is presented in the figure below.



In Appendix A, one can find a detailed description of the LOGTRAN.

4.2 Statements Translation Routine (STATRAN)

The sequence of steps performed by STATRAN are summarized as follows:

1) Ask the user to give a name for the procedure he is creating and then calls the routine PFOPEN (an Interpreter's routine) to open the file needed for the PROC file.

2) Reads the user statement which will come in a format similar to the TSOS Command Language. If the statement is a Local Name Assembler command (LOCNAME), STATRAN processes each Local name parameter and enters it into a Local Names Control Block (LNCB). This LNCB is a special internal format which is used only by STATRAN itself in substituting the Local Names appearing in a future statement.

The LNCB is a variable length block of up to ten elements of the following format:

A_S	L_S	A_V	L_V
4 bytes	4 bytes	4 bytes	4 bytes

where

A_S = Address of a synonym or Local Name

L_S = Length of that synonym or Local Name

A_V = Address of the value, i.e. description, to be substituted for the specific Local Name, and

L_V = Length of the value

After this is done, it reads the next statement, searches for local names and substitutes the appropriate values if encountered. Then it goes to the following step, which is the same when no Local Name command is present.

3) STATRAN sets up the internal format and branches to the PFRECADD routine (another Interpreter's routine) which will make the addition of the 'record' or statement to the file that was created in step (1).

The internal format is a parameter list that always must be passed to each Interpreter's routines (collectively called PFROUT).

The format for this parameter list is:

0 - 71	Reg. Save Area for Assembler Register
72 - 143	Reg. Save Area for PFROUT Register
144 - 147	Reserve Area (for use by Assembler)
149 - 157	Procedure File Name
159 - 165	Reserve Area for use by PFROUT
167 - 175	Action or Operation Name
177 - 415	Parameter list for the action name.

Steps 2) and 3) are repeated until the user enters the statement

END*

which terminates a normal procedure.

4) Calls the routine PFCLOSE (a third Interpreter's routine) to close the now completed PROC file.

The following is an example of an interactive session between the user and the Assembler (the responses from the Assembler are underlined):

```
/EXEC PROC
 ENTER PROCEDURE NAME
*SEARCH
 BEGIN
*FLIST
*RETRIEVE FILE1,C'TOPIC=POPULATION "AND" CITY=PHILADELPHIA
 "AND" YEAR=1969,11
*CALC
*STATICS
*END*
/
```

The above statements cause the STATRAN to set up a PROC file.

The user supplies the name SEARCH which becomes the name of the PROC file. He then wants a list of the program in his file; this is done by execution of the program FLIST. He then uses the RETRIEVE COMMAND to request the retrieval of one record from FILE1. After the retrieval is performed with the Logical Expression parameter of the RETRIEVE command, the user wants a "calculator" program to be executed, which is done by issuing the CALC command of the regular TSOS system. Finally it executes a program of his own called STATICS and closes the PROC file by issuing the END* Assembler command.

Appendix B gives more details of STATRAN together with its flowchart. Examples of typical interactive sessions with the Assembler are found in Appendix D.

4.3 Format of the Command Elements

The functions described below were considered to be an essential part of the Extended Data Management System. Since these functions should be quickly and easily available to the user, they were implemented as commands. This section discussed the format of these commands.

4.3.1 The Formal Parameter Command

The Formal Parameter (FPARAM) command allows the user to supply the actual values for formal parameters within a PROC file which has previously been created.

The format for this command is

Operation	Operands
FPARAM	PROC file name, new PROC file name, formal parameter 1, value 1 [,formal parameter 2, value 2] [,formal parameter 3, value 3]

The operand has 3 required positional operands plus 2 optional positional operands.

PROC file name - This operand specifies the name of the procedure file created by the user. This operand is required.

new PROC file name - This operand specifies the name of a temporary PROC file to be created which will contain the actual values in place of the formal parameters. This operand is required.

formal parameter 1 - This operand specifies the first formal parameter. This operand is required.

value 1 - This operand specifies the actual value associated with the first formal parameter. This operand is required.

formal parameter 2, value 2
formal parameter 3, value 3 - These four optional parameters specify another possible one or

two formal parameters which are a part of the PROC file and the actual values which they are to assume.

The user can replace more than three formal parameters by repeated use of FPARAM.

In creating the PROC file the user should place a dollar sign ($) in front of each formal parameter he wants to substitute at a later time. The formal parameter must be ten characters or less in length. The actual value may of course be more than ten characters. Examples of the use of this command can be found in [3].

4.3.2 The Retrieve Command

The Retrieve Command (RETRIEVE) allows the user to retrieve records from the files of the Extended Data Management System by logical expressions of keywords associated with the records.

The format for this command is:

Operation	Operands
RETRIEVE	file name, C'logical expression' [,no. of records][,output spec.][,label]

where

file name - the name of the file from which the record(s) are to be retrieved. This operand is required.

logical expression - The logical expression of keywords for the retrieval request. This operand is required.

no. of records - This operand specifies the maximum number of records which the user desires to have retrieved. The default case is all records.

label - This operand specifies a label for referring to this retrieve command in the continue command (Sect. 4.3.3).

output spec. - This is a parameter to specify the type of output specification desired for the retrieved records.

Appendix E shows an actual retrieval from a data base.

4.3.3 The Continue Command

The Continue command (CONTINUE) allows the user to continue a retrieval process which had previously been initiated by the RETRIEVE command.

The format for this command is

Operation	Operands
CONTINUE	label [,no. of records]

The parameter descriptions are:

label - This operand is the name in the label field of a RETRIEVE command which is to be continued. This operand is required.

no. of records - This operand specifies the number of records to be retrieved. The operand is optional. The default case is the same number as appeared in the RETRIEVE command.

4.3.4 The Restore Command

After a retrieval this command will allow the user to make addition, deletion or replacement of specific lines in the retrieved record and restore it to the file. The format for this command is:

Operation	Operands
RESTORE	file name, record number

The description of the operands is:

file name - This operand specifies the name of the file from which a retrieval has been performed and in which addition, deletion or replacement of lines are going to be performed.

record number - A code for the retrieved record in which the restore operation will be done.

After the user issues the RESTORE command, the system will respond with an asterisk(*) which will indicate that it is waiting for data from the user. He then uses the following input format for his data (underlined asterisks are printed by the system):

a) Replacement of line:

* line number attribute=value*

where

line number - The current line he wants to replace.

attribute - The new attribute he wants in the line.

value - The new value he wants in the line.

b) for deletion

* line number 1 - line number 2

The command then will erase from line number 1 up to and including line number 2 as specified.

c) for addition

The format is exactly as in a). Since all existing line numbers are multiples of 100, the user must issue a new line number that lies between the line numbers of the old ones and into which the new line is to be inserted. For example, if in an old record the line numbers are as follows:

line	text
⋮	⋮
300	TOPIC = LOGIC
400	YEAR = 1964

Then the specification of a line as follows will result in the line being inserted between lines 300 and 400.

305 AUTHOR=SMITH

The new record will look as follows with line numbers adjusted:

line	text
⋮	⋮
300	TOPIC = LOGIC
400	AUTHOR = SMITH
500	YEAR = 1964

To terminate the update and initiate the restore operations, the user enters the following command:

END

It is assumed that if the user sends a line number n, all lines with numbers less than n are not going to be modified and hence they will be a part of the new record. Then it will be impossible to modify a line number less than n once it has been sent, unless he issues the RESTORE command again.

Basically the RESTORE servicing routines perform a retrieval based on the 'record number' of the particular record the user wishes to update. After the record has been retrieved, it is transformed from 'internal format' into 'core format'. The routine then sets up a buffer in which the new record is stored with the new or deleted lines. An update routine (UPDATCOM) is called to make the addition of the new record and another routine (DELREC) performs the deletion of the old one.

A flow chart for the servicing routine of this command is found in Appendix C.

CHAPTER 5

CONCLUSIONS

We have provided the Extended Data Management Facility with a simple but powerful interactive query language schema. The language schema enables the user to define new problem solving procedures and to make use of previously defined procedures.

A statement of a procedure enables the user to bring in a program by name for execution, and to provide the running program with parameters.

The user can perform the retrieval or other operations on records of some files. He is merely required to specify the set of records with a logical expression of their keywords, and to include the logical expression in the statement of the operation.

We think that we have developed the basic command and query language for a "generalized" system. By generalized system we mean that tools and elements needed for future growth into more sophisticated language schemata are present. These are:

1. standard methods for the addition of new commands and their command processing routines into the Facility [6];

2. procedures of statements for immediately or later execution;

3. statements for comprising standard TSOS commands, new commands of the Facility and users' programs and their associated parameters;

4. parameters for including logical expression of keywords.

All the implementation has been done having "modularity" in mind. By modularity we mean that each element of the language, and in many cases the programs that comprise the command and query language are "stand-alone" elements or programs. That is, the programs and elements can be used for some specific purpose without going through the normal

Assembler-Interpreter flow by any user (e.g. the FPARAM command of the Interpreter and the LOGTRAN SVC implemented in the Assembler). This in turn will make it easy to make improvements in the future tendent toward a more sophisticated language or system.

Some of the improvements we can foresee for the Assembler are given below.

As was explained before (Section 4.1) the Logical Expression Translating routine (LOGTRAN) accepts a logical expression in "external" disjunctive normal form. This restriction can be tedious for the user in writing a description. For example, given the following description,

(AUTHOR=SMITH 'AND' YEAR=1964) 'OR' (AUTHOR=JONES 'AND' YEAR=1964)
'OR' (AUTHOR=MINSKY 'AND' YEAR=1964)

we would like to shorten it as follows:

(AUTHOR=SMITH 'OR' JONES 'OR' MINSKY) 'AND' YEAR=1964

In other words what we need is a 'pre-processor' which will transform any logical expression in "free external" form into DNF format.

To make the Extended Data Management Facility a custom-tailored system for a user, we believe the above mentioned basic tools and language schemata would enable the user to build more sophisticated language pre-processors and general purpose command and query language repertoire.

BIBLIOGRAPHY

1. Interim Technical Report, "An Integrated Information Storage, Retrieval, and Dissemination Facility," Moore School of Electrical Engineering, University of Pennsylvania, June 1, 1969.

2. Manola, F., "An Extended Data Management Facility for a General Purpose Time Sharing System," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, work in progress.

3. McDonald, J.N., "A Command and Query Language Interpreter for an Extended Data Management System," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August 1970.

4. Ets. A.R., "The File Searching, Record Validating and Record Formatting Functions of the Supervisor for an Extended Data Management Facility," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August 1970.

5. Hirsch, J., "The Access Control and Retrieval Optimization Functions of the Supervisor for an Extended Data Management Facility," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, August 1970.

6. Russell, R.N., "A Manual on Adding New Commands and SVC's to TSOS," Moore School of Electrical Engineering, University of Pennsylvania, September 1969.

7. Horton, M., "Reading, Writing, Creating and Updating Records and Files in a Generalized File Structure," M.Sc. Thesis, The Moore School of Electrical Engineering, University of Pennsylvania, work in progress.

8. Desiato, B., "Directory Constructing and Decoding in a Generalized File Structure," M.Sc. Thesis, The Moore School of Electrical

Engineering, University of Pennsylvania, August 1970.

APPENDIX A

ROUTINE LOGTRAN

LOGTRAN - Logical Expressions translating routine. LOGTRAN function is to translate an input string which is a logical expression of keywords in disjunctive normal form (DNF) into an internal format to be processed by an interpreter.

A.1 Entry Points

LOGTRAN has two entry points. LTRNENTA is the entry point for system routines while LTRNENTB is the SVC entrance.

A.2 Exit Points

LOGTRAN has two normal exit points plus an exit point in case of system error. SYSEXIT will return control when LOGTRAN is called by a system routine. SVEXIT will do it when called as an SVC. ERREXIT will terminate execution in case of a system abnormal task termination.

A.3 Input Parameter List

The address of the input parameter list (LTRPARM) must be in Register 1. If the calling routine is a system program, Register 13 must contain the address of a save area.

Name	Length	Content
LTRPARM	DSECT	
LTRADD	F	Address of the logical expression
LTRLNGTH	F	Length of the logical expression

A.4 LOGTRAN Output

LOGTRAN output is one block of information composed of a header of 5 bytes with control information plus two sub-blocks: a Description Control Block (DCB) and a Key Information Buffer (KIB). The complete block is of variable length.

LOGTRAN Output Specifications

Header

1 byte	Internal code, not used by LOGTRAN
2 bytes	Length of KIB and DCB
2 bytes	Length of DCB

Description Control Block

Bytes	4	1	2	3	3	5
	A_1	C_1	L_{a_1}	L_{fv_1}	$L_{\ell v_1}$	F_1'
	A_2	C_2	L_{a_2}	L_{fv_2}	$L_{\ell v_2}$	F_2'
	A_3	.	.	.	.	F_3'
	v	End of elementary conjunct				
	A_4	.	.	.	.	F_4'
	A_5	.	.	.	.	F_5'

Key Information Buffer

Attribute	Value	Attribute	Val
ue	Attribute	Value	...
....	....	....	...

Figure A.1

Description of Control Block Areas for Figure A.1.

A_i is a pointer to the beginning of the i^{th} keyword that is stored in the Key Information Buffer.

C_i is the control code that indicates the relation between the attribute and the value.

L_{a_i} is the length of the i^{th} attribute.

L_{fv_i} is the length of the first value of the i^{th} keyword.

L_{lv_i} is the length of the last value of the i^{th} keyword.

F_i' is the pointer to the beginning of a list of format numbers associated with the attribute.

A.5 Return Codes

All return codes can be found in the right-most byte of Register 15 and they are listed below by hexadecimal digits.

X'00'	Everything OK
X'02'	Improper relation
X'06'	Improper connective
X'08'	Wrong multiple value
X'0A'	Missing value

Whenever an error occurs (right-most byte of Register 15 non zero), Register 1 is loaded with the address of a parameter area with the appropriate message.

A.6 Register Conventions

The registers in LOGTRAN are assigned in the following manner:

Register	Utilization
0	Not used
1	Miscellaneous use
2	Miscellaneous use
3	Counter of elements of DCB.
4	Input index.
5	Base for ELEM DSECT. DCB element format.
6	Switch to indicate type of calling routine (SVC or system routine).
7	Relative address of keywords.
8	Length of logical expression.
9	Index for KIB.
10	Base register for LOGTRAN.
11	Base register for BLKDSECT work area.
12	Not used.
13	Address of save area in calling system routine.
14	Return address in calling system routine.
15	Error codes.

A.7 Internal Work Area

BLKDSECT is the internal work area used by the LOGTRAN routine. It has the following format:

Name	Length	Content
BLKDSECT	DSECT	Work area.
INTCODE	CL1	Not used by LOGTRAN.
LENOUT	CL2	Length of KIB and DCB.
LENCLB	CL2	Length of DCB.
CLBLCK	10CL18	Allocation for DCB.

Name	Length	Content
OUTPUT	200C	Allocation for KIB.
LENGTHA	F	Length of attribute.
LENGTHV1	F	Length of first value.
LENGTHV2	F	Length of second value.
ADDR	F	Temporary storage.
LENOUTF	F	Temporary storage.
RELADD	F	Temporary storage.
LENCLBF	F	Temporary storage.
EIASTK	F	Address of EIA stack area.
ELEM	DSECT	* DCB Format.
ADDRESS1	CL4	Relative address of attribute.
RELATION	CL1	Relation code.
LENPAT	CL2	Length of attribute.
LENPVL1	CL3	Length of first value.
LENPVL2	CL3	Length of second value.
FONUPT	CL5	Format number pointer.

A.8 Internal Codes

The internal codes in the LOGTRAN routine are listed below by hexadecimal digits:

RELATION - relation between Attribute and Value

a) Codes for normal Attribute-Value search

Internal Code	User Code	Relation
X'F0'	'NQ'	NOT EQUAL or negation
X'F1'	'EQ', =	EQUAL
X'F2'	'LT', <	LESS THAN
X'F3'	'LE'	LESS OR EQUAL THAN

Internal Code	User Code	Relation
X'F4'	'GT', >	GREATER THAN
X'F5'	'GE'	GREATER OR EQUAL THAN
X'F6'	'TO'	Multiple value search [value 1] 'TO' [value 2]

b) Codes for search on value only.

Internal Code	User Code	Relation
X'E0'	'NE'	NOT EQUAL or negation
X'E1'	'EQ', =	EQUAL
X'E2'	'LT', <	LESS THAN
X'E3'	'LE'	LESS OR EQUAL THAN
X'E4'	'GT', >	GREATER THAN
X'E5'	'GE'	GREATER OR EQUAL THAN
X'E6'	'TO'	Multiple value search [value 1] 'TO' [value 2]

The end of a conjunct in the logical expression is indicated by:

Internal Code	User Code
X'4E'	'OR', ∨, +

This code is found in the RELATION or code portion of an element of the DCB. The remaining of this element of the DCB is filled with zeros.

Each time the user code 'AND' (&) is found in the logical expression, a new element is added to the DCB and it is filled with control information from the keyword that follows that code.

A.9 Flowchart

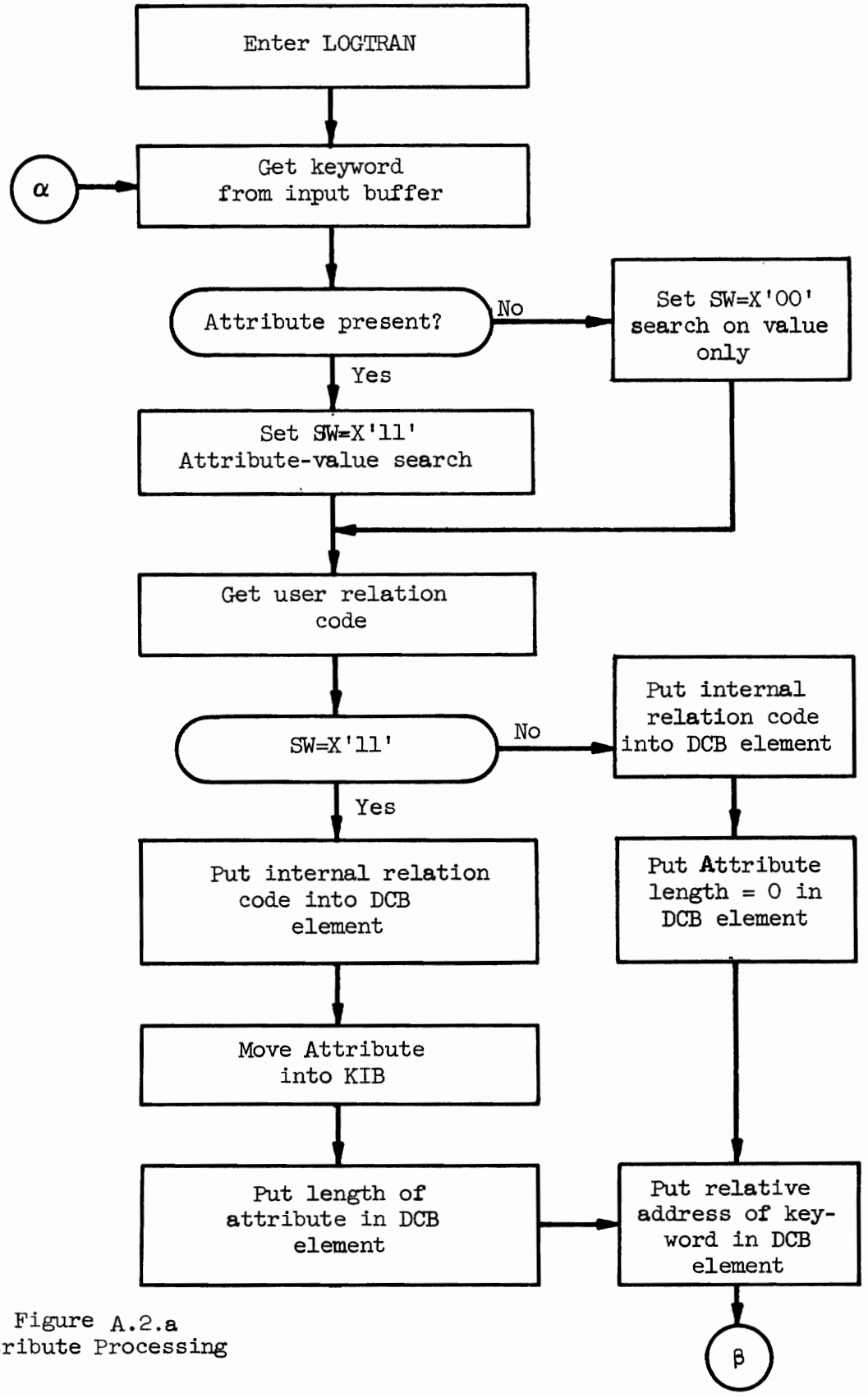


Figure A.2.a
Attribute Processing

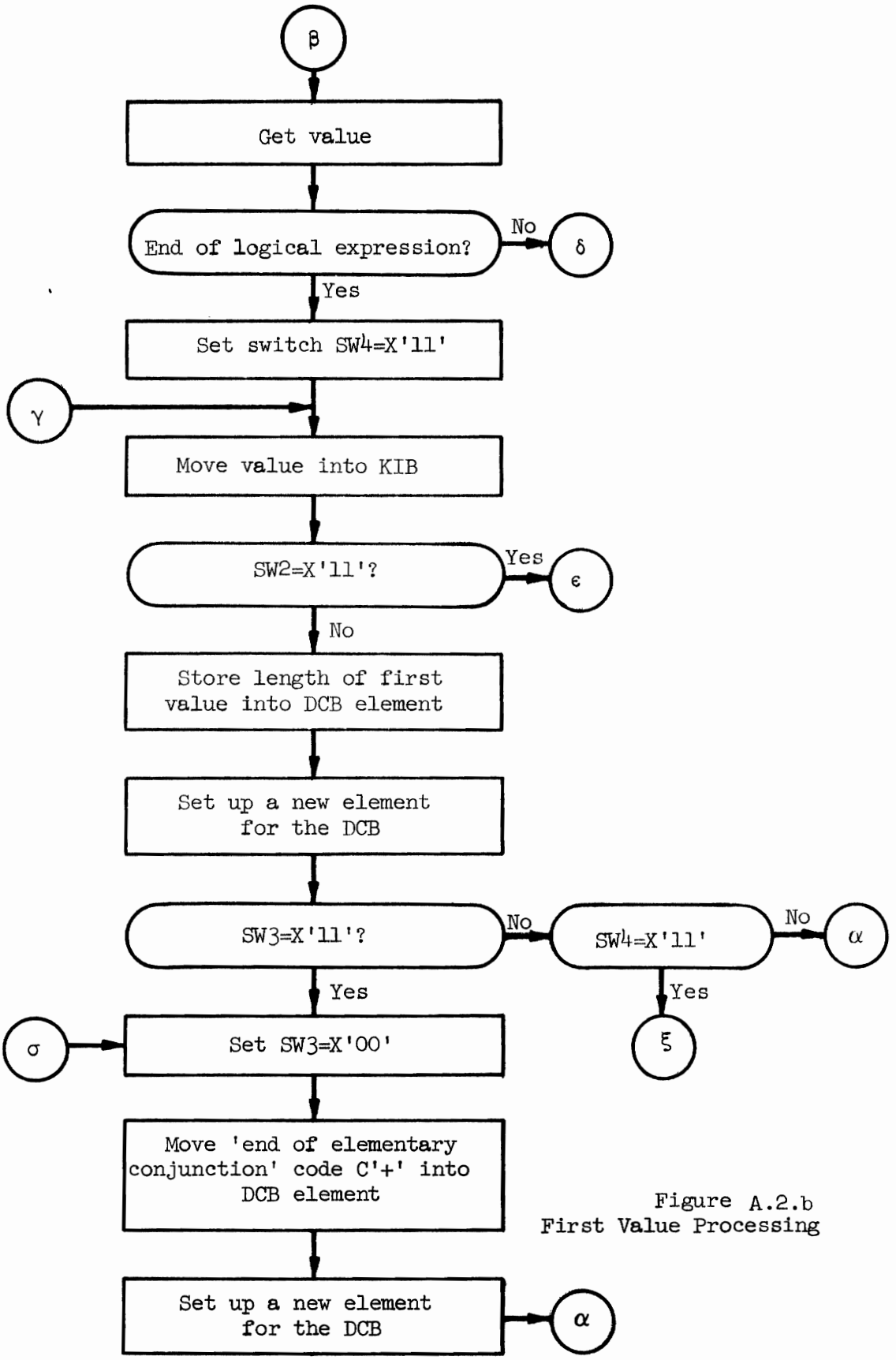


Figure A.2.b
First Value Processing

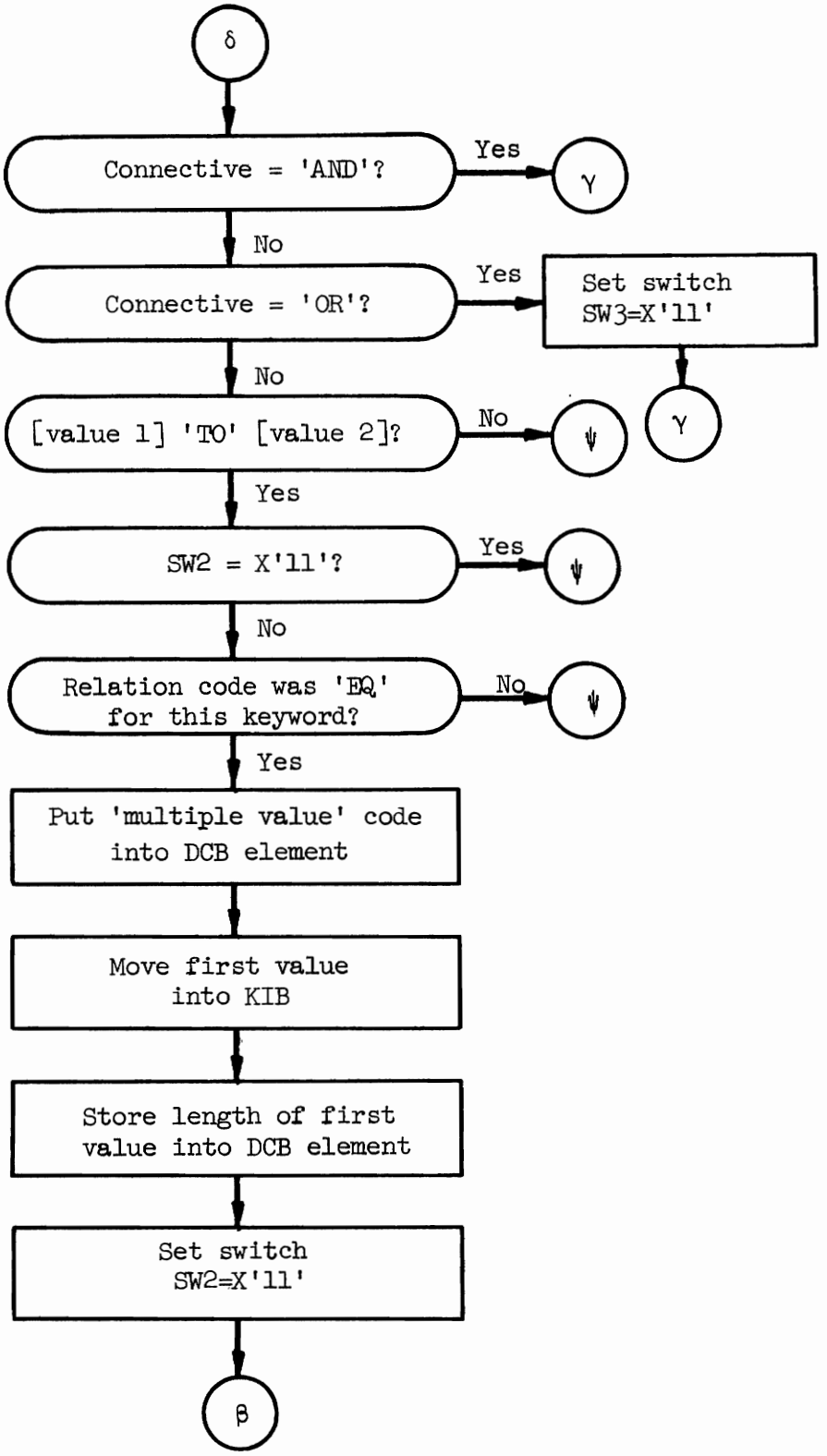


Figure A.2.c

Multiple Value Processing

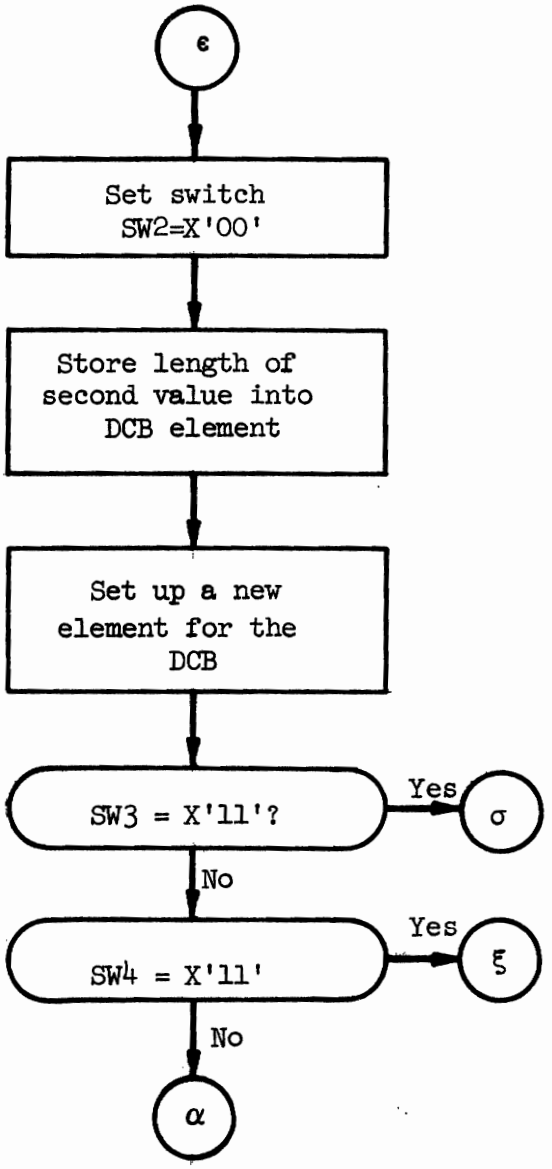


Figure A.2.d

Second Value Processing

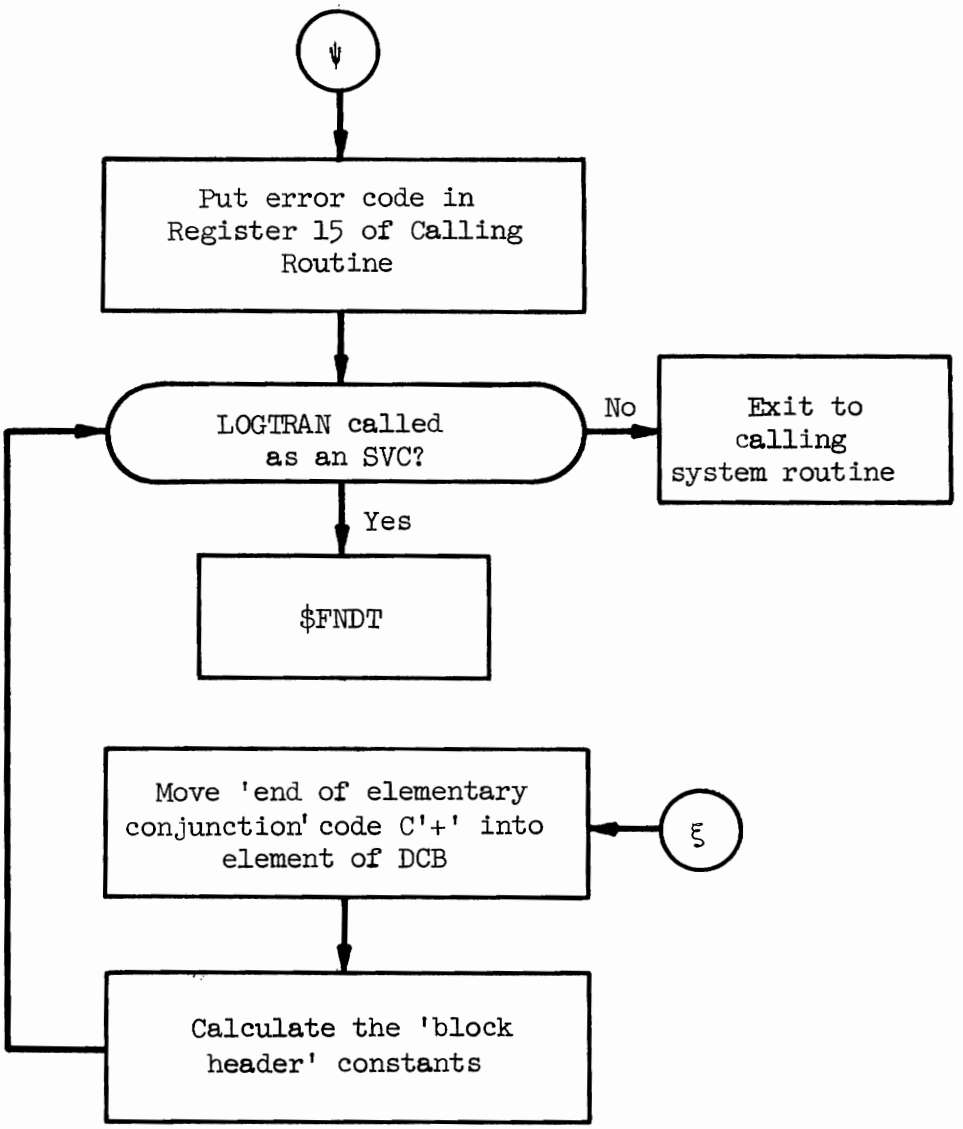


Figure A.2.e

LOGTRAN Exits

APPENDIX B

ROUTINE STATRAN

STATRAN routine is the Assembler section of the PROC routine. PROC routine function is to create a Procedure file of TSOS commands which can be executable later on by means of the /DO [procedure name] command. The actual creation of this procedure is carried out by the Interpreter. The STATRAN routine checks the input syntax and sets up the appropriate parameters and internal format (if a statement is accepted) to be passed to the Interpreter. The STATRAN routine also processes the local names when they are encountered.

B.1 Entry Point

STATRAN has only one entry point at PROC. PROC is also the name of the CSECT and should be used as [program name] when the user wants to execute it.

B.2 Exit Point

There is only one exit point for this routine. It begins at ENDTRAN where control is returned to the user in the command mode.

B.3 External Subroutine Calls

There are three external subroutines that are called by STATRAN. These are:

Name of Entry Point	Function
PFOPEN	Procedure file open routine
PFRECADD	Procedure file record addition routine
PFCLOSE	Procedure file close routine

These Interpreter's routines are called collectively PFROUT. Detailed specifications on PFROUT can be found in [3].

STATRAN sets up the internal format. This internal format is nothing but a parameter list that always must be passed to each PFROUT routine.

Following is the DSECT for this parameter list:

Name	Length	Content
PCFPM	DSECT	
SAVE	CL72	Register Save Area for Assembler Registers
	CL72	Register Save Area for PFROUT Registers
FILNAME	OCL14	
	CL4	Reserved
PFNAME	10C	Procedure file name
	F	Reserved area for PFROUT
	F	Reserved area for PFROUT
OPRTNM	CL10	Action or Operation name
PARAMT	CL240	Parameter list for the Action name

B.4 Register Conventions

The registers are assigned in the following manner:

Register	Utilization
0	Not used.
1	Temporary storage.
2	Temporary storage.
3	Temporary storage.
4	Input index.
5	Counter
6	Base Register for PFDS, work area for STATRAN
7	Address of Save Area
8	Address of PCFPM procedure file parameters and temporary storage

Register	Utilization
9	Counter of synonyms of Local Names
10	Base Register for STATRAN
11	Base Register for SYNON, DSECT used in processing Local Names.
12	Pointer for processing Local Names
13	Not used.
14	Return address in STATRAN.
15	Called subroutines address.

B.5 Internal Work Area

PFDS is the internal work area used by the STATRAN routine. It contains the parameter list (PCFPM) that is passed to the PFROUT routines. The work area has the following format:

Name	Length	Content
PFDS	DSECT	
INPUTO	OCL254	
CONTROL	CL4	Type V record header
INPUT	250C	Input record
BK	CL1 CL1	Delimiter
PAREA	OCL12	Parameter area for RDATA macro.
EDIT	CL1	EDIT option
ADD	CL3	Address of type V record
LENGTH	CL4	Length of record
RES2	CL1	Reserved
ERRAD	CL3	Address of error routine
ADDTEM	F	Temporary storage.
ERRADD1	F	Temporary storage.

Name	Length	Content
LNBUFF	250C	Buffer for processing statements with Local Names
INPUT1	250C	Buffer for input statements with Local Names
SW	CL1	Switch indicator for Local Names
NUMBEL	F	Number of synonyms
CKBK	40F	Control Block area for Local Names
PCFPM	OF	Procedure file parameter list
SAVE	CL72	Register Save Area for Assembler
	CL72	Register Save Area for PFROUT
FILNAME	OCL14 CL4	Reserved
PFNAM	10C	Procedure file name
	F	Reserved for PFROUT
	F	Reserved for PFROUT
OPRTNM	CL10	Action or Operation name
PARAMT	CL240	Parameter list for the action name
SYNON	DSECT	Used to create a control block for Local Names
ATTADD	F	Address of synonym or attribute
LENAT	F	Length of attribute
ADDVAL	F	Address of value for a Local Name
VALEN	F	Length of Value

B.6 Flowchart

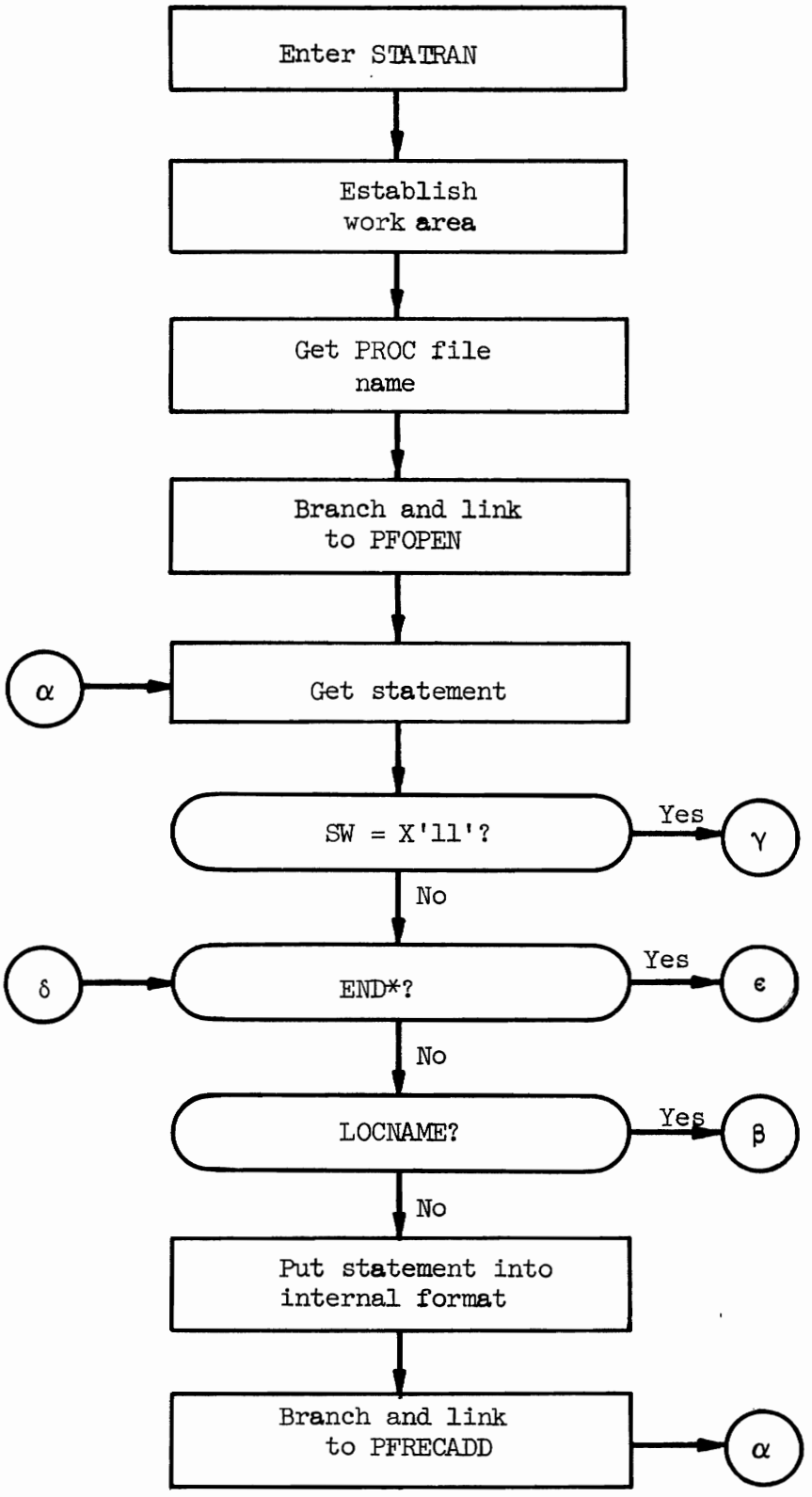


Figure B.1.a
Statements Processing

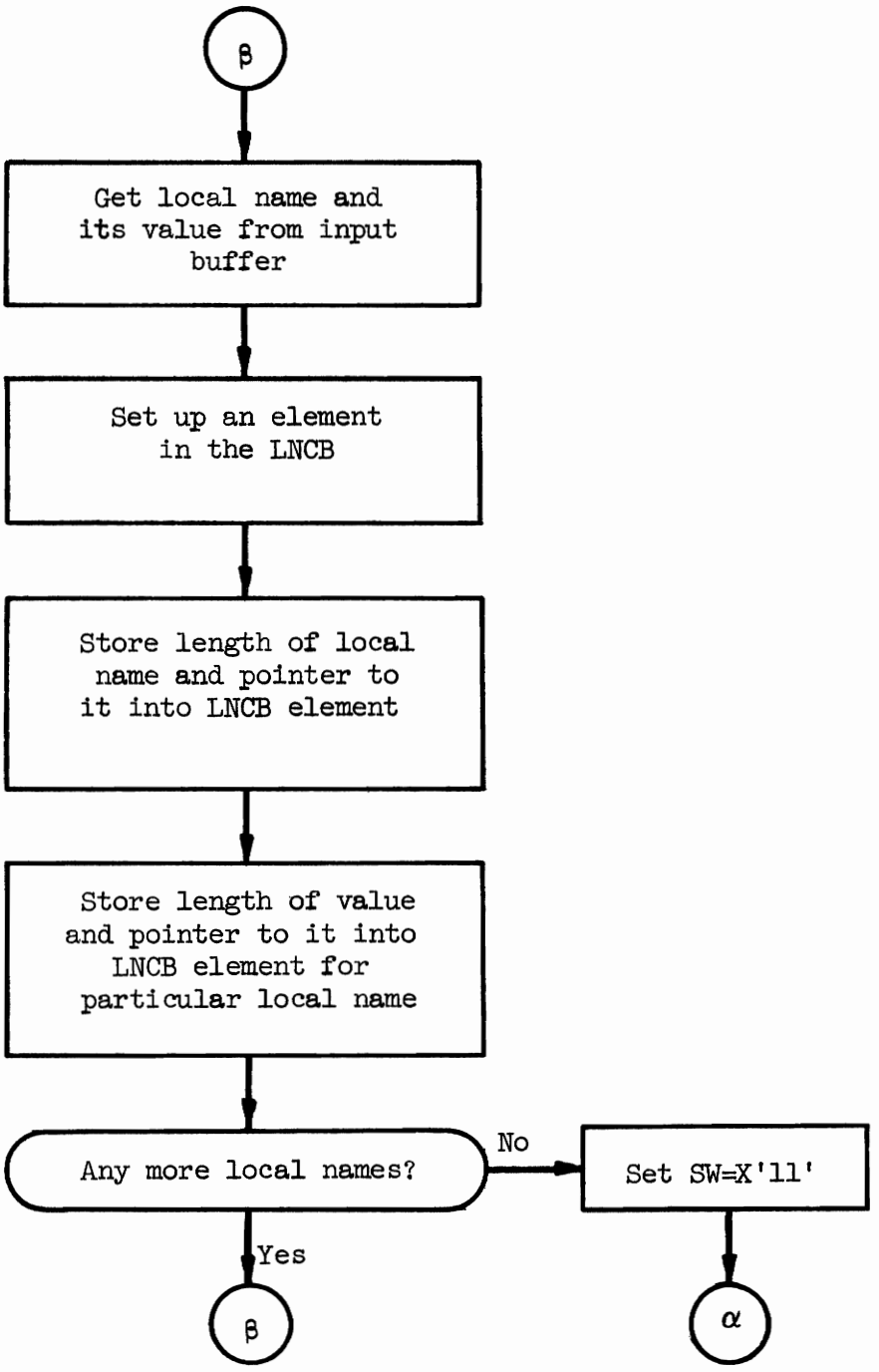


Figure B.1.b

LOCNAME Statement Processing

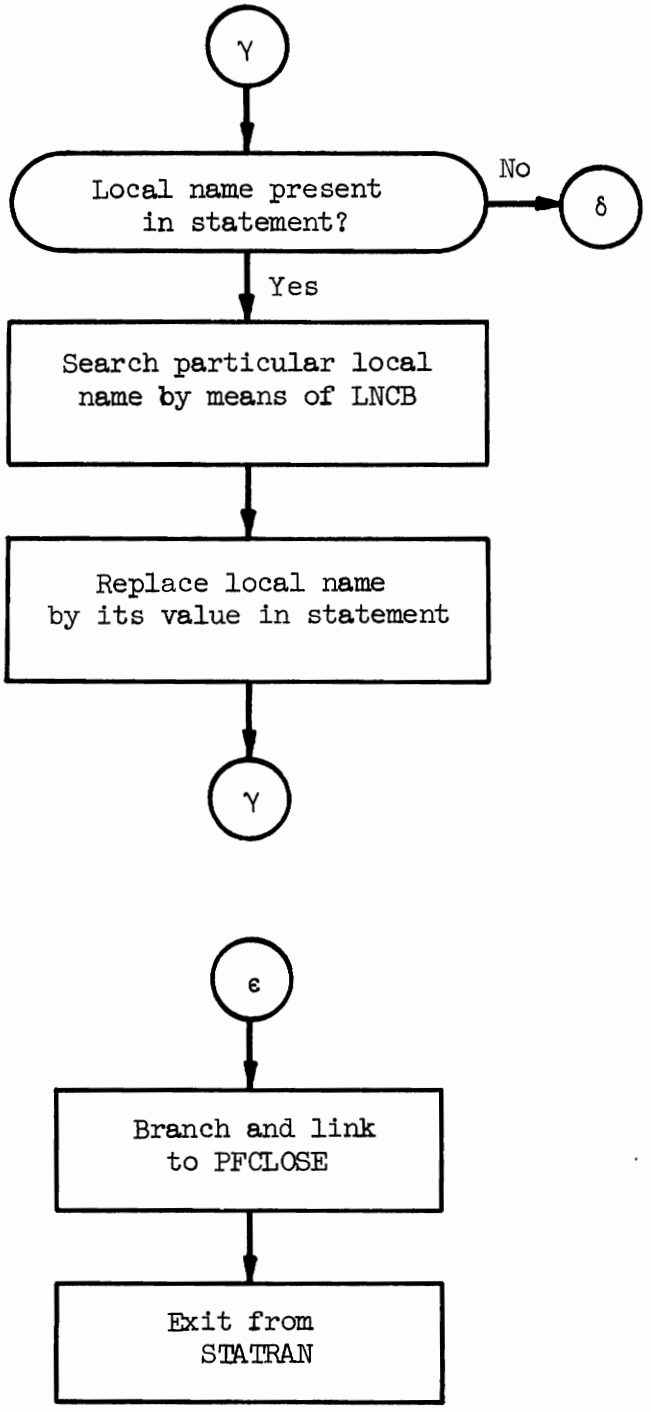


Figure B.1.c

Local Names Replacement and Exit from STATRAN

APPENDIX C

RESTORE COMMAND SERVICING ROUTINE

After a retrieval this command allows the user to make addition or deletion of specific lines in a particular record and restore it to its position in the file.

RESTREC is the Command Servicing routine for the RESTORE command.

C.1 Input

Execution of this routine is initiated by the Terminal Command Processor (TCP). The TCP passes the following information to RESTREC.

1) Register 1 contains the address of the Interpretative Scanner Processor (ISP) parameter list.

2) Register 13 contains the address of the register save area to be used by RESTREC.

3) Register 0 contains the address of an additional save area which may be used by the called program when branching to ISP and any other subprogram.

After calling on ISP, RESTREC has the following parameter list available to it. (This list is the output of ISP.)

Name	Length	Content
ISPOUTPT	DSECT OF	
RESFILNM	CL54	File Name
RESRECNM	F	Record Number

C.2 External Subroutine Calls

There are 6 external subroutines that RESTREC call upon. These subroutines together with their entry points and input parameter lists are given below:

C.2.1 SSBCHECK

This routine will output the address of the SSB (Service Status Block) in Register 1.

From the SSB block, RESTREC will get the FCB address and RFB address, which are needed later on.

Entry Point

SSBCHECK

Input Parameter List

Name	Length	Content
PARSSB	DSECT	
TASKADD	F	Address of Task Control Block
FILENGTH	H	Length of file name
FILNAME	CL54	File name

Service Status Block (SSB)

Name	Length	Content
SSB	DSECT	
SSBHDR	OF	SSB HEADER
SSBUAI	F	Address of user authority item
SSBFIF	F	Address of FCB for FIF
SSBHLEN	*-SSBHOR	Length of SSB HEADER
SSBTXTD	DSECT	
SSBTXT	OF	SSB Test
SSBFNAM	14F	2 byte length, 54 bytes file name
SSBCL	F	Control information
SSBLFIB	*-SSBTXT	Length from start of SSB text to FIB entry
SSBFIB	F	Address of FIB for file name

Name	Length	Content
SSBFCB	F	Address of FCB for file name
SSBOTBIN	OC	X'FF' indicates description present
SSBOTAB	F	ADDR of user description block
SSBCREC	F	ADDR of core format record
SSBFSB	F	Address of FSB block
SSBTLG	CL1	Control information for pointer
SSBPTR	AL3	PTR to next SSB block
SSBTLEN	*-SSBTXT	Length of SSB text.

C.2.2 RETRREC

This routine will perform a retrieval of a particular record in internal format based upon the ISAM key of that particular record.

Entry Point

RETRREC

Input Parameter List

Name	Length	Content
RESTFCB	F	FCB address
REINREAD	F	Buffer address
RESISAM	CL5	ISAM key
REINRSIZ	CL3	Size of buffer

C.2.3 COREFMT

This routine input is the record just retrieved in internal format and outputs it in core format.

Entry Point

COREFMT

Input Parameter List

Name	Length	Content
COINREAD	F	Address of internal record
CORRFB	F	Address of RFB
CORFMADD	F	Address of core format buffer

C.2.4 DELREC

This routine will delete a record. It is used in RESTREC in order to delete the old record just retrieved in internal form.

Entry Point

DELREC

Input Parameter List

Name	Length	Content
RECADD	F	Address of record in internal form
FCBADDR	F	Address of FCB
	D	Reserved for use by DELREC routine

C.2.5 RESTORE

The actual update of the record is performed by this routine. It gets the input data from the user with the information of the particular line(s) he wishes to update or delete, checks the syntax of the input statements and helps the user in issuing the right statements. Finally it sets up a new updated record in core format.

Entry Point

RESTORE

Input Parameter List

Name	Length	Content
NWRECADD	F	Address of new record
OLDRECAD	F	Address of old COREFMT record

Figure C.1 shows the Core Format Record specification.

C.2.6 UPDATCOM

This routine restores the new record to its position in file.

Entry Point

UPDATCOM

Input Parameter List

Name	Length	Content
FILADDR	F	Address of 54 bytes area with file name
ROCADDR	F	Address of core format record

For detailed specifications on some of these external routines, the reader is referred to [7].

C.3 Flowchart

3 bytes	Size of Record	
5 bytes	Reference number, unpacked	
1 byte	Control information	
3 bytes	Length of attr.-value entry	ATTRIBUTE VALUE ENTRY
1 byte	Control information	
1 byte	Number of Directory Lists	
2 bytes	Length of attribute	
variable	Attribute	
3 bytes	Length of Value	
variable	Value	
3 bytes	Length of attr.-value entry	
1 byte	Control Information	
1 byte	Number of Directory Lists	
2 bytes	Length of attribute	
variable	Attribute	
3 bytes	Length of Value	
variable	Value	
	. . . . .	

Note: "Number of Directory Lists" field is used for those attribute-value pairs which are used as keywords when file characteristics allow a variable number of directory lists. Field is ignored for other attribute-value pairs.

The length specified in the 3 byte "Size of Record" entry includes the 9 byte Header size.

Figure C.1

Core Format Record Specification

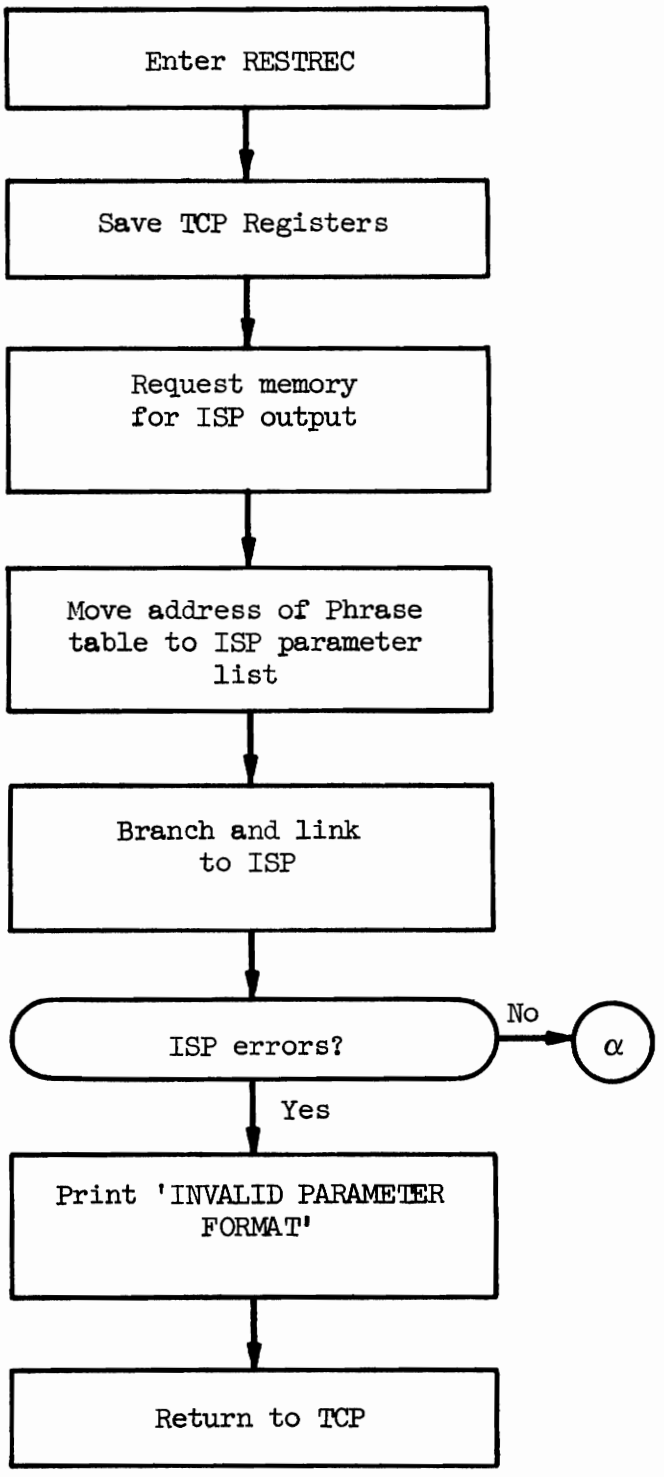


Figure C.2.a

RESTREC Routine

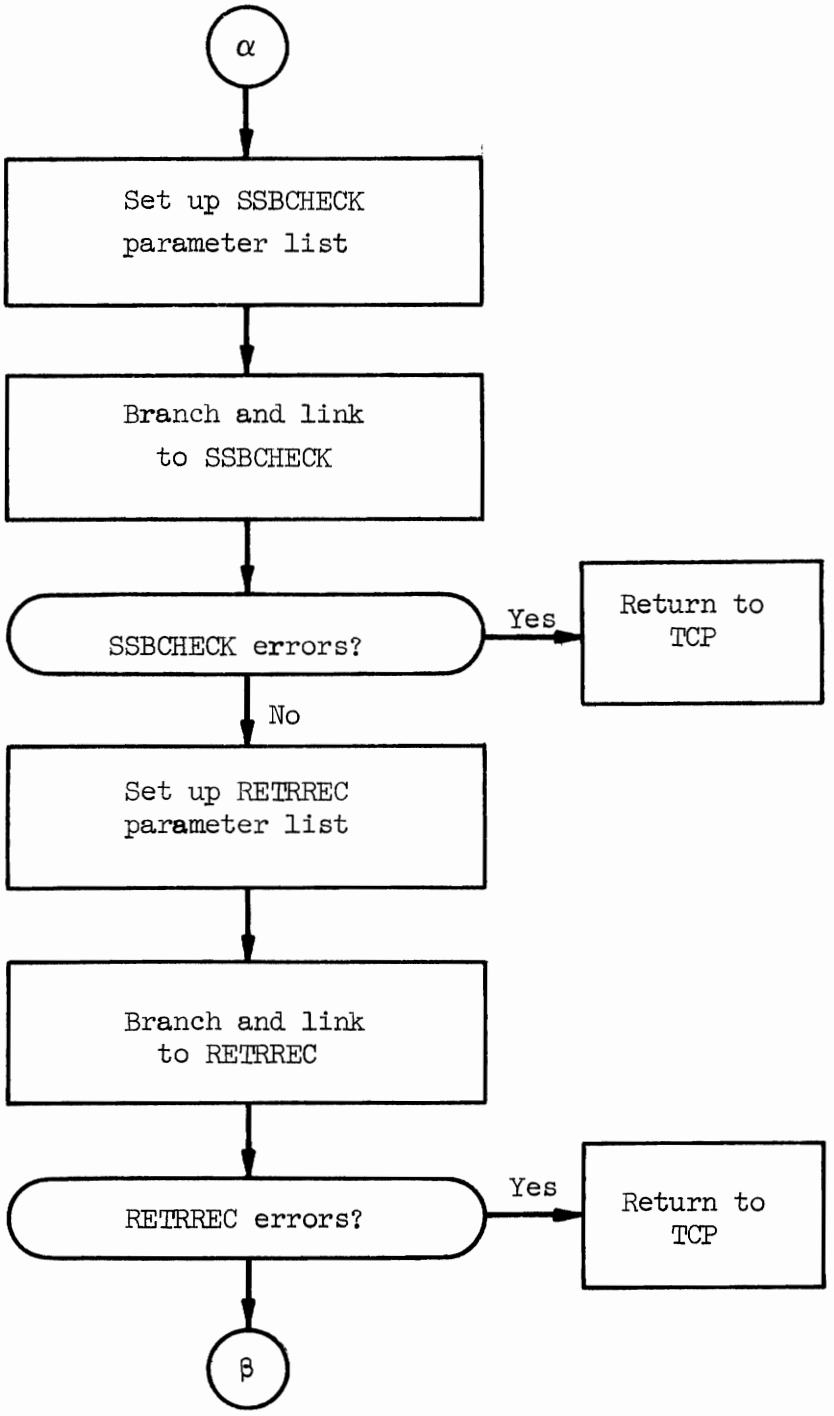


Figure C.2.b

RESTREC Routine (cont.)

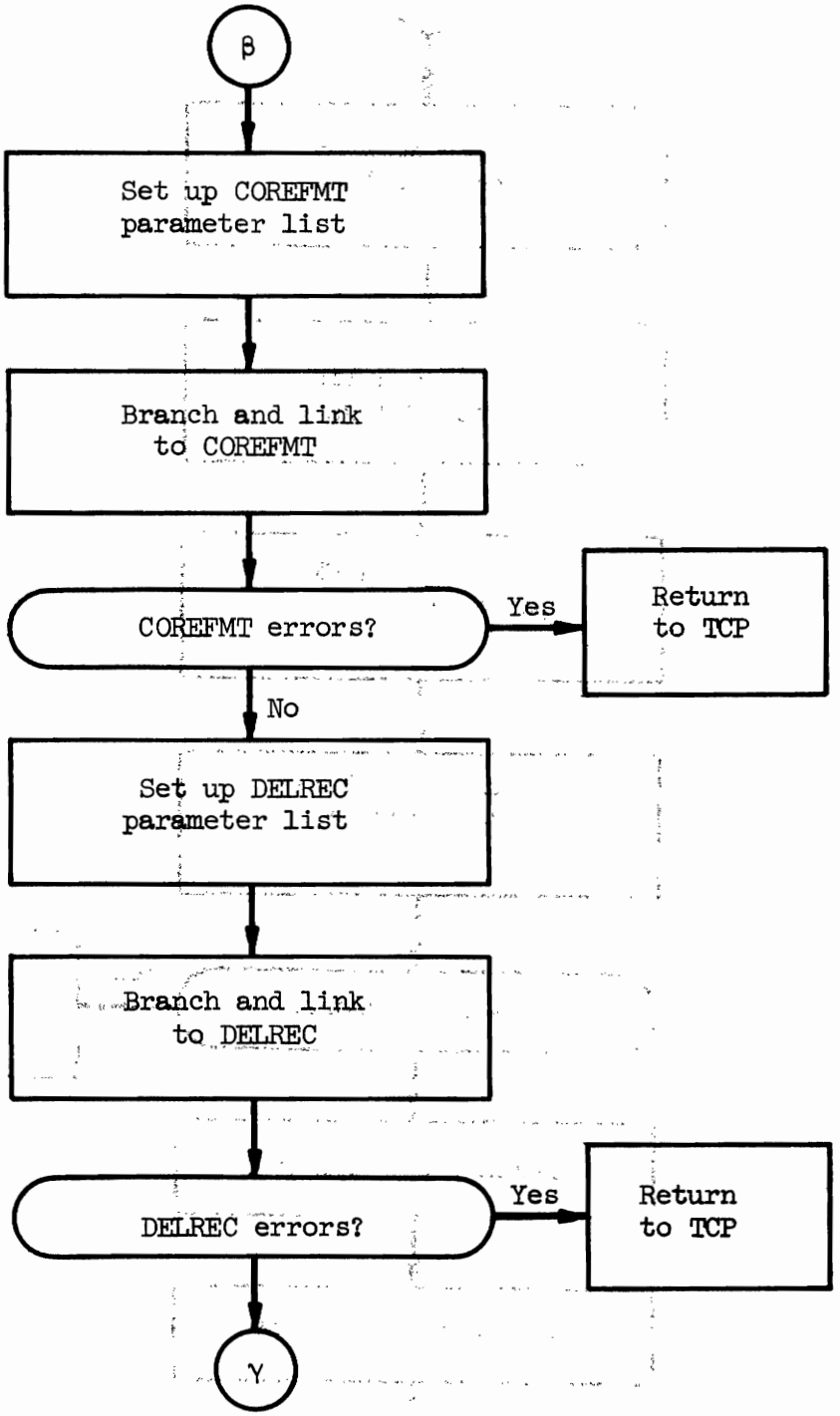


Figure C.2.c

RESTREC Routine (cont.)

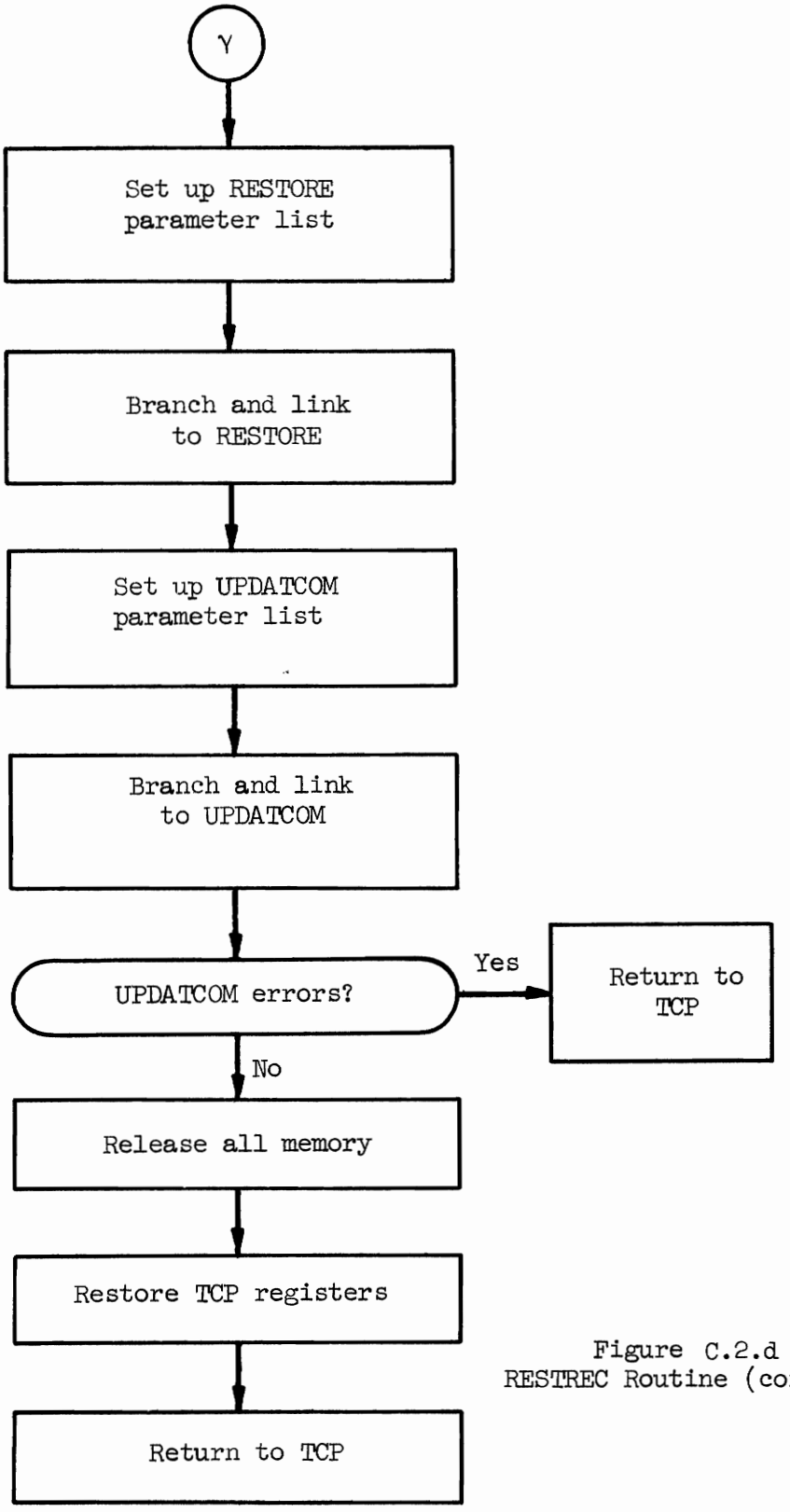


Figure C.2.d
RESTREC Routine (cont.)

APPENDIX D

EXAMPLES OF PROCEDURE CREATIONS

```
ZB001 PLEASE LOGON.
/LOGON GANA
ZB002 LOGON ACCEPTED AT 1523 ON 09/15/70, TSN 8666 ASSIGNED.
/EXEC PROC
ZL001 PROGRAM LOADING
 ENTER PROCEDURE NAME
*EXAMPLE1
 BEGIN
*EXEC $DESIATO.LOGTRAN
*CALC
*FLIST
*END*
 END OF PROCEDURE
/DO EXAMPLE1
Z/PROC C
Z/EXEC $DESIATO.LOGTRAN
ZL001 PROGRAM LOADING
*TOPIC=LOGIC & YEAR=1964,
   REQUEST IS BEING PROCESSED
  ENTERED RCDCHK
  ENTERED RCDCHK
  ENTERED RCDCHK

000100 CODE NUMBER = BENNJ641
000200 AUTHOR = BENNETT, J. M.
000300 AUTHOR = EASTON, W. B.
000400 AUTHOR = GUARD, J. R.
000500 AUTHOR = LOVEMAN, D. B.
000600 AUTHOR = MOTT, T. H. JR.
000700 TITLE = SEMI-AUTOMATED MATHEMATICS-SAM IV
000800 KEY PHRASES = SEMI-AUTOMATED MATHEMATICS
000900 KEY PHRASES = SAM IV
001000 COMPANY = APPLIED LOGIC CORPORATION
001100 COMPANY ADDRESS = PRINCETON, N. J.
001200 DOCUMENT NUMBER = AFCRL NO. 64-827 (CONTRACT NO. AF19(628)-3250
, SCIENTIFIC REPORT NO. 3)
001300 MONTH = OCTOBER
001400 YEAR = 1964
001500 PAGES = 85
001600 TOPIC = LOGIC
001700 TOPIC = THEORM
001800 ABSTRACT = THIS REPORT IS THE FOURTH ON THE SUBJECT OF THE TITL
E, AND IS PRECEDED BY MOTT630, GUARJ640, AND BENNJ640.  NONALGORITHMI
C PROOF PROCEDURES ARE USED, AND THE HUMAN SUPPLIES INFORMATION TO ASS
IST IN THE PROOF.  A LIMITED CLASS OF MATHEMATICAL STATEMENTS CAN BE H
ANDLED. THIS PROGRAM IS ON THE IBM 7040.

   REQUEST PROCESSING COMPLETED
Z/CALC
*SQRT(678)
ZG = +26.0384331
*X=SQRT(430)**1.6
ZX = +127.871796
*Y=34.5**2.4
ZY = +4906.43259
*X+Y
ZG = +5034.30439
*END
Z/EXEC FLIST
ZL001 PROGRAM LOADING
 PROC 0005
 RESTORE 0004
 LOGTRAN 0004
 COREFMT 0003
 MULTRET 0012
 TEMP 0012
 JORGE 0024
 MACPROC 0003
 SLAM5 0004
 SLAM6 0003
 EXAMPLE1 0002
Z/ENDP
/LOGOFF
ZB003 LOGOFF AT 1532 ON 09/15/70, FOR TSN 8666.
ZB014 CPU TIME USED: 0006.8513 SECONDS.
```

Example D.1
Typical Procedure Creation Followed by
Immediate Execution

```
%B001 PLEASE LOGON.
/LOGON GANA
%B002 LOGON ACCEPTED AT 1144 ON 09/18/70, TSN 8939 ASSIGNED.
/EXEC PROC
%L001 PROGRAM LOADING
 ENTER PROCEDURE NAME
*EXAMPLE2
 BEGIN
*LOCNAME #AA :=$HORTON.MULTTES1 #BB :=C'YEAR=1964' ###
*RETRIEVE #AA,#BB,1
*RESTORE #AA,05
*SAMPLE #BB
*FSTATUS #AA
*END*
 END OF PROCEDURE
/EXEC (EDIT)
%L001 DYNAMIC LOADER INVOKED
 VERS. 0009 OF FILE EDITOR READY
*O T N
 OPENED T AS NEW V-TYPE FILE.
*GET EXAMPLE2
*P 0 $
 /PROC C
 /RETRIEVE $HORTON.MULTTES1,C'YEAR=1964',1
 /RESTORE $HORTON.MULTTES1,05
 /SAMPLE C'YEAR=1964'
 /FSTATUS $HORTON.MULTTES1
 /ENDP
*H
/LOGOFF
%B003 LOGOFF AT 1200 ON 09/18/70, FOR TSN 8939.
%B014 CPU TIME USED: 0008.9271 SECONDS.
```

Example D.2

Use of Local Names in Creating a
Procedure for Later Execution

APPENDIX E

RETRIEVE SESSION EXAMPLE

```
ZB001 PLEASE LOGON.
/LOGON GANA
ZB002 LOGON ACCEPTED AT 1534 ON 09/11/70, TSN 8472 ASSIGNED.
/RETRIEVE $HORTON.MULTTES1,C'YEAR=1964''TO''1968''AND''TOPIC=LOGIC',2
   REQUEST IS BEING PROCESSED
  ENTERED RCDCHK
  ENTERED RCDCHK

 RECORD NO. 000014
 000100 CODE NUMBER = COOPD661
 000200 AUTHOR = COOPER, D. C.
 000300 TITLE = THEOREM-PROVING IN COMPUTERS
 000400 KEY PHRASES = THEOREM-PROVING
 000500 BOOK = ADVANCES IN PROGRAMMING AND NON-NUMERICAL COMPUTATION
 000600 PUBLISHER = PERGAMON PRESS
 000700 YEAR = 1966
 000800 PAGE LIMITS = 155-182
 000900 TOPIC = LOGIC
 001000 TOPIC = THEORM
 001100 ABSTRACT = THIS PAPER IS BASED ON SOME LECTURES GIVEN AT A
        SUMMER COURSE IN ENGLAND IN 1963.  IT CONTAINS A GENERAL SURVEY
        OF SOME OF THE BASIC ISSUES AND WORK DONE, AS WELL AS SPECIFIC
        DETAILS ABOUT THE VARIOUS ALGORITHMS FOR PROVING THEOREMS IN
        LOGIC OF THE CITED WORKERS.  THERE IS A LENGTHY DESCRIPTION OF
        HERBRAND'S THEOREM.
  ENTERED RCDCHK

 RECORD NO. 000002
 000100 CODE NUMBER = BENNJ641
 000200 AUTHOR = BENNETT, J. H.
 000300 AUTHOR = EASTON, W. B.
 000400 AUTHOR = GUARD, J. R.
 000500 AUTHOR = LOVEMAN, D. B.
 000600 AUTHOR = MOTT, T. H. JR.
 000700 TITLE = SEMI-AUTOMATED MATHEMATICS-SAM IV
 000800 KEY PHRASES = SEMI-AUTOMATED MATHEMATICS
 000900 KEY PHRASES = SAM IV
 001000 COMPANY = APPLIED LOGIC CORPORATION
 001100 COMPANY ADDRESS = PRINCETON, N. J.
 001200 DOCUMENT NUMBER = AFCRL NO. 64-827 (CONTRACT NO. AF19(628)-3250,
        SCIENTIFIC REPORT NO. 3)
 001300 MONTH = OCTOBER
 001400 YEAR = 1964
 001500 PAGES = 85
 001600 TOPIC = LOGIC
 001700 TOPIC = THEORM
 001800 ABSTRACT = THIS REPORT IS THE FOURTH ON THE SUBJECT OF THE
        TITLE, AND IS PRECEDED BY MOTTT630, GUARJ640, AND BENNJ640.
        NONALGORITHMIC PROOF PROCEDURES ARE USED, AND THE HUMAN SUPPLIES
        INFORMATION TO ASSIST IN THE PROOF.  A LIMITED CLASS OF
        MATHEMATICAL STATEMENTS CAN BE HANDLED. THIS PROGRAM IS ON THE
        IBM 7040.

   REQUEST PROCESSING COMPLETED
/LOGOFF
ZB003 LOGOFF AT 1539 ON 09/11/70, FOR TSN 8472.
ZB014 CPU TIME USED: 0001.8480 SECONDS.
```